THE UNIVERSITY
of EDINBURGH

THE UNIVERSITY *of* EDINBURGH

A New Local Search in the Space of Infeasible Solutions Framework for the Routing of Vehicles

Mona Hamid

Doctor of Philosophy

The University of Edinburgh

2018

Abstract

Combinatorial optimisation problems (COPs) have been at the origin of the design of many optimal and heuristic solution frameworks such as branch-and-bound algorithms, branch-and-cut algorithms, classical local search methods, metaheuristics, and hyperheuristics.

This thesis proposes a refined generic and parametrised infeasible local search (GPILS) algorithm for solving COPs and customises it to solve the traveling salesman problem (TSP), for illustration purposes. In addition, a rule-based heuristic is proposed to initialise infeasible local search, referred to as the parameterised infeasible heuristic (PIH), which allows the analyst to have some control over the features of the infeasible solution he/she might want to start the infeasible search with. A recursive infeasible neighbourhood search (RINS) as well as a generic patching procedure to search the infeasible space are also proposed. These procedures are designed in a generic manner, so they can be adapted to any choice of parameters of the GPILS, where the set of parameters, in fact for simplicity, refers to set of parameters, components, criteria and rules.

Furthermore, a hyperheuristic framework is proposed for optimizing the parameters of GPILS referred to as HH-GPILS. Experiments have been run for both sequential (i.e. simulated annealing, variable neighbourhood search, and tabu search) and parallel hyperheuristics (i.e., genetic algorithms / GAs) to empirically assess the performance of the proposed HH-GPILS in solving TSP using instances from the TSPLIB. Empirical results suggest that HH-GPILS delivers an outstanding performance.

Finally, an offline learning mechanism is proposed as a seeding technique to improve the performance and speed of the proposed parallel HH-GPILS. The proposed offline learning mechanism makes use of a knowledge-base to keep track of the best performing chromosomes and their scores. Empirical results suggest that this learning mechanism is a promising technique to initialise the GA's population.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Mona Hamid

Dedicated to my parents and siblings

for their love, endless support

and encouragement

Acknowledgements

First and foremost, I would like to thank my parents for providing this opportunity. None of this would have been possible without their sacrifice, motivation and inspiration. Also, I would like to thank my siblings with unfailing support and continuous encouragement throughout my entire life. This accomplishment would not have been possible without them.

Importantly, I would like to thank my supervisor Professor Jamal Ouenniche. I would like to show my gratitude to Professor Ouenniche for sharing his knowledge and expertise with me during the course of this research. I am grateful to him for his exceptional support, motivation and assistance with the algorithms and techniques. I sincerely appreciate his expertise that has greatly assisted my research and his belief in my work.

I thank all my friends, especially Johanna, Angela, Adriana, Louise, Ben, Elizabeth and Ja for their intellectual and emotional support. I greatly value their friendship. They are my extended family, who have aided and encouraged me throughout this endeavour and made my time here more fun. I want to highlight the tremendous quality service of Susan Keating and thank her for all the instances in which her assistance helped me along the way.

Thank you.

Mona Hamid

Contents

List of Figures

List of Tables

1. Introduction

The area of combinatorial optimisation arises from research in computer science (Lawler, 1976). The studied combinatorial optimisation problems (COPs) are very diverse. Several solution methodologies have been proposed to solve these problems; namely, exact solution approaches and heuristic solution frameworks.

Exact solution approaches (section 2.3) solve COPs to optimality. However, these approaches are in general computationally intensive and their efficiency depends on the choice of the bounding schema used to prune nodes in the search tree to avoid exploring branches with no potential to deliver an optimal solution. Solution methods within this category can be categorised as branch-and-bound algorithms, cutting plane algorithms, and their hybrids as branch-and-cut algorithms.

On the other hand, heuristic solution approaches (section 2.4) can, in general, obtain optimal or near-optimal solutions quicker; e.g. construction heuristics, local search-based methods and metaheuristics (section 2.5). Although the optimality of the solutions delivered with heuristics cannot be proved, they can often find a good quality solution to large problems in a reasonable time. Considering the advantages and disadvantages of the exact and heuristic solution approaches one can combine ideas taken from each of these methods and develop a stronger hybrid.

Notice however that heuristic solution approaches are often tailor-made to a specific problem. Moreover, they might produce good quality solutions for some instances of the problem, but not all. Thus, one has to either investigate the problem and its instances structure thoroughly or design a higher-level method to automate the choice of which heuristic or combination of heuristics and their parameters to use (Burke et al., 2009). Such high-level methodologies are called hyperheuristics (section 2.6).

Most of the heuristic methods mentioned above start with a feasible solution and only search in their feasible neighbourhood to find a better solution. One disadvantage of these methods is that they limit the search to feasible neighbourhood areas. However,

there are methods that allow infeasibilities while searching for a better solution. These methods penalise infeasibilities to force the search toward the feasible neighbourhoods. One disadvantage of these methods is that they allow for a limited degree of infeasibility. However, recently Ouenniche and his collaborators (Ouenniche et al., 2017) proposed a local search method, called dual local search (DLS), which starts with an infeasible solution and explores its infeasible neighbourhood for a better neighbour until the search reaches a feasible solution – for more details see section 2.7. DLS integrates the design features of exact methods (branch-and-bound) into heuristic methods (local search). Since DLS is a parametric method, one can design a hyperheuristic framework to optimise the choice of its parameters and components.

Thus, we can categorise heuristic solution approaches into search methods that search in the space of feasible solutions, feasible-infeasible solutions and infeasible solutions. In the rest of this thesis we shall call these methods feasible methodologies, feasible-infeasible methodologies and infeasible methodologies. Note that feasible methodologies are also called primal methodologies, thus we use these terms interchangeably.

This thesis refines and extends DLS proposed by Ouenniche et al. (2017) and proposes a generic and parameterised infeasible local search (GPILS). The proposed framework is generic since it can be used to solve any COP; however, for illustration purposes, GPILS is customised to solve the travelling salesman problem. Since GPILS is a parameterised framework, each set of parameters leads to a different GPILS procedure. Thus, it should be viewed as a collection of GPILS procedures each corresponding to a different set of parameters, which can be chosen either by the analyst or by an automated process. We propose a generic hyperheuristic framework to automate the optimisation of the choice of parameters of GPILS, referred to as HH-GPILS, where the focus is on metaheuristic-based high-level frameworks. Moreover, an offline learning mechanism is proposed to speed up HH-GPILS, which reuses the previously generated set of parameters for the unseen (new) problem instances.

1.1. Goal and scope of the research

The aim of this research is to investigate the possibilities of a different avenue to solve COPs. We explored the infeasible search space by proposing a new generic and parameterised infeasible space search framework to solve COPs as well as an automated procedure to optimise the choice of its parameters. The goals of this thesis may be summarised as follows:

1. *Investigate the possibility of heuristically searching the infeasible solution space and progressing toward the feasible space.*

   In chapter 3, we propose a new generic and parameterised local search framework that operates in the space of infeasible solutions and, for illustration purposes, customise it to solve the travelling salesman problem and discuss its implementation decisions.
2. *Automate and optimise the choice of the parameters of the proposed framework.*

   In order to optimise the choice of the parameters of our local search methodology, we propose a hyperheuristic framework. We experiment with both sequential high-level mechanisms (chapter 4), namely simulated annealing, tabu search, variable neighbourhood search as well as new hybrids of these metaheuristics, and a parallel high-level mechanism (chapter 5), namely genetic algorithm.
3. *Investigate the potential of reusing the automatically generated parameters.*

   In chapter 5, we propose an offline learning mechanism for the parallel high-level framework to improve its search strategy. The proposed offline learning makes use of a knowledge base to keep track of the best performing set of parameters in the past and reuses them when facing new problem instances.

The novelty of this research lies principally in the search strategy to explore the infeasible space to find the optimal or near optimal solution, GPILS. To the best of our knowledge, little research has been done in the exploration of the infeasible space. The empirical investigation shows that this search strategy has a promising future.

1.2. Contributions

Firstly, we proposed a generic parameterised infeasible local search (GPILS) that starts the search in the infeasible space and continues the search, through the infeasible space, toward the feasible space with the option of continuing the search in the feasible space. We explained the rationale behind the design of the proposed GPILS and for illustration purposes we customised it for solving the TSP.

Furthermore, we automated the choice of the parameters of GPILS using a hyperheuristic framework, referred to as HH-GPILS. We proposed several sequential and parallel metaheuristic-based high-level methods to search the parameter space of the GPILS. Note that the proposed HH-GPILS can be used as high-level framework in any hyperheuristic that aims to automate the choice of parameters.

Finally, we proposed a new offline learning mechanism to improve the performance of the HH-GPILS. We developed a knowledge-based system that is used to keep the best performing sets of parameters and their scores. Furthermore, a reward/penalty mechanism is proposed to update the score of each set. Note that these scores are used as criteria for their entrance and survival in the knowledge-base.

1.3. Thesis structure

Chapter 2: Literature Review

This chapter surveys the literature on exact solution approaches and heuristic solution frameworks to solve the TSP, as well as hyperheuristics.

Chapter 3: Generic Parameterised Infeasible Local Search Framework

In chapter 3, the proposed generic parameterised infeasible local search framework (GPILS) is presented, and the rationale behind the infeasible search methodology is discussed. Then, we customised GPILS for the TSP, for illustration purposes. Finally, the implementation decisions and parameters of GPILS are explained.

Chapter 4: A Sequential Hyperheuristic Framework for GPILS

In this chapter, a generic high-level framework to automate the choice of the parameters of GPILS is presented. Later, the proposed sequential high-level frameworks, namely simulated annealing, tabu search, variable neighbourhood search and their hybrid are explained in detail. Moreover, new neighbourhood structures are proposed to search the parameter space of GPILS. Lastly, we analysed the performance of the proposed sequential high-level frameworks using the proposed neighbourhood structures.

Chapter 5: A Parallel Hyperheuristic Framework for GPILS

A parallel high-level framework, namely genetic algorithm (GA), to optimise the parameters of GPILS is presented in chapter 5. In this framework, the proposed GA makes use of indirect chromosome representation where each chromosome is encoded as a vector of parameters of GPILS.

Moreover, an offline learning mechanism is proposed to improve the performance and speed up the parallel high-level framework for GPILS. The proposed learning mechanism that makes use of a knowledge-based system, referred to as chromosome base (CB), to keep track of well-performing chromosomes and their score. A reward-based mechanism is used to update scores of each chromosome from the CB and compute the score of the new chromosomes. Later, the CB is updated by replacing a number of previous chromosomes in the CB with the new chromosomes.

Chapter 6: Conclusion

In this chapter, we presented the concluding remarks.

Appendices

Appendix A: Tour construction heuristics

Several primal Tour construction heuristics, such as nearest neighbour procedure; Clarke and Wright savings procedures; insertion procedures; Christofides heuristic; nearest merger procedures; path merging procedures, are explained in more detail.

Appendix B: Cooling strategies

Some of the cooling schedules proposed for the simulated annealing are presented, such as Aarts and Van Laarhoven (1985, 1987), Lundy and Mees (1986), Huang et al. (1986), Triki et al. (2005), Dowsland (1993) and Azizi and Zolfaghari (2004).

Appendix C: Acceptance function

Several of the existing acceptance functions used in simulated annealing are presented, namely, Aarts and Van Laarhoven (1985a, 1987), Lundy and Mees (1986), Huang et al. (1986), Triki et al. (2005), Dowsland (1993) and Azizi and Zolfaghari (2004).

Appendix D: Neighbourhood change strategies

In this appendix, the classification of the neighbourhood change strategies by Hansen et al. (2016) are presented. They classified neighbourhood change strategies into sequential, cyclic, pipe, and skewed neighbourhood change strategy.

Appendix E: GA's selection mechanisms

In this section, the GA's selection mechanisms are presented, namely ordinal selection, proportional selection, ranking selection, steady-state selection.

Appendix F: Crossover techniques

Several crossover techniques have been used in the GA's search, some of these techniques, such as simple or one-point crossover, multi-point crossover, uniform crossover and three parents' crossover, are presented in this appendix.

Appendix G: Hyperheuristics classification and categories

Several hyperheuristic classifications and categories have been proposed such as Soubeiga (2003), Bai (2005) and Ross (2005), Bader-El-Den and Poli (2007), Chakhlevitch and Cowling (2008), Burke et al. (2009, 2010, and 2013). These classifications are presented in this section.

Appendix H: Learning mechanisms

Several learning mechanism have been proposed such as choice function (Cowling et al., 2001; Chen et al., 2016), reinforcement learning (Nareyek, 2003; Ozcan et al.,

2010; Chen et al., 2016), learning classifier system (Holland and Reitman, 1978; Ross et al., 2002; Marín-Blázquez and Schulenburg, 2007) and case-based reasoning (Petrovic and Qu, 2002, Burke et al. 2002, 2004, 2006). These learning mechanisms are presented in this section.

Appendix I: $\mathcal{K}$-Means clustering

In this thesis, we used $\mathcal{K}$-Means clustering method (Jain, 2010) to exploit the structure of TSP instances. In this section, we explained this procedure in more details.

Appendix J: RINS function

Exploring the infeasible space by the proposed GPILS given a set of parameters requires exploring a single or several combinations of the set. Thus, to implement an efficient and generic code, we proposed a recursive design for the infeasible neighbourhood search, referred to as recursive infeasible neighbourhood search function (RINS), see Chapter 3. In this appendix, in addition to details of RINS, a summary of recursive function is presented.

2. Literature Review

In this chapter, the relevant scientific literature will be provided, particularly mathematical formulations of the TSP, their properties and relaxations, descriptions of the methodologies and state-of-the-art techniques used in this study are presented

2.1. Mathematical formulation

TSP is the prototype problem in combinatorial optimisation, introduced in 1930 (Applegate et al. 1998). Because of its applications in different fields such as vehicle routing and scheduling, computer wiring, job sequencing, drilling of circuit boards, and order picking in warehouses, many solution methodologies have been proposed to solve TSP.

In graph theory, TSP is defined on a complete weighted graph, say $G = (V, E)$ where V is set of vertices or nodes and E is set of arcs. The vertices represent the cities and the arcs represent the links between pairs of cities. Let x_{ij} be a binary decision variable which is equal to one if arc (i, j) is in the optimal solution (i.e., included in the optimal TSP route) and zero otherwise; and $C = (c_{ij})$ be a non-negative distance matrix. TSP is symmetric if for all (i, j), $c_{ij} = c_{ji}$ and asymmetric if for some or all (i, j), $c_{ij} \neq c_{ji}$.

Several formulations have been proposed for the TSP. These formulations can be classified as conventional formulations (Dantzig, Fulkerson and Johnson; 1954), sequential formulations (Miller, Tucker and Zemlin; 1960), time-staged formulations (Vajda, 1961; Fox, Gavish and Graves, 1980), and flow-based formulations (Gavish and Graves, 1978; Finke, Claus and Gunn, 1984; Lucena, 1986; Wong, 1980; Claus, 1984; Langevin, 1988; Loulou, 1988). All these formulations are assignment problem (AP) relaxation-based with different subtour breaking constraints. The difference between conventional and sequential formulations lie in the nature of their subtour breaking constraints. On the other hand, time staged formulations introduce a time

stage, say t, and ensure that when a node is visited at stage t, it is left at stage $t+1$. As to flow-based formulations, as the name suggests, flow constraints are used to prevent the formation of subtours. Hereafter, we shall present the above mentioned formulations, but first we present the general TSP formulation of as follows:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \quad 1$$

Subject to:

$$\sum_{j=1}^{n} x_{ij} = 1,\ i \in V \quad 2$$

$$\sum_{i=1}^{n} x_{ij} = 1,\ \ j \in V \quad 3$$

Subtour elimination constraints 4

$$x_{ij} = 0\ or\ 1\ ,\ \ i,j \in V \quad 5$$

where (1) is the objective function which minimises the total cost of the TSP tour; (2) and (3) are the degree constraints; (4) is the subtour elimination constraints and (5) is the binary constraints. Hereafter we shall present common subtour elimination constraints proposed in the literature.

The most cited mathematical formulation of TSP in the literature is the conventional formulation by Dantzig et al. (1954). They proposed the following subtour elimination constraints for the TSP:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1,\ \ \forall S \subset V,\ 2 \leq |S| \leq n-2. \quad 6$$

These constraints can be stated differently as connectivity constraints:

$$\sum_{i \in S} \sum_{j \in V-S} x_{ij} \geq 1,\ \ \forall S \subset V, S \neq \emptyset \quad 7$$

Note that this TSP formulation consists of $n(n-1)$ binary variables, $2n$ degree constraints and $2^n - 2n - 2$ subtour elimination constraints. Thus, for a problem with 6 nodes, there are 30 binary variables and in total 62 constraints. An alternative is the sequential formulation of Miller, Tucker and Zimlin (1960), which reduces number of subtour breaking constraints by introducing extra decision variables u_i that represent the sequence in which city i is visited, and introduced the following constraints known as the MTZ subtour breaking constraints:

$$u_i - u_j + (n-1)x_{ij} \leq n-2,\ \forall i,j \in \{2,\dots,n\}; i \neq j \quad (8)$$

$$1 \leq u_i \leq n-1,\ \forall i \quad (9)$$

Later, Desrochers and Laporte (1991) strenghend MTZ constraints by adding an extra term

$$u_i - u_j + (n-1)x_{ij} + (n-3)x_{ji} \leq n-2, i,j \in \{2,\dots,n\}; i \neq j \quad (10)$$

On the other hand, time-stage formulations introduce binary decision variables y_{ij}^t , which are equal to one if arc (i,j) is travelled at stage t and zero otherwise, and the following constraints:

$$\sum_{i=1}^n \sum_{j=1}^n \sum_{t=1}^n y_{ij}^t = n, \quad (11)$$

$$\sum_{j=1}^n \sum_{t=2}^n t y_{ij}^t - \sum_k^n \sum_{t=2}^n t y_{kj}^t = 1,\ \ i = 2,\dots,n \quad (12)$$

$$x_{ij} - \sum_{i=1}^n \sum_{t=1}^n t y_{ij}^t = 0, \forall i,j \in N, i \neq j \quad (13)$$

$$y_{ij}^t = 0\ or\ 1\ ,\ \ i,j,t = 1,\dots,n \quad (14)$$

$$y_{il}^t = 0\ \ \forall t \neq n, y_{ij}^t = 0\ \ \forall t \neq 1,\ \ y_{ij}^l = 0\ \ \forall i \neq 1, i \neq j \quad (15)$$

where constraint (11) ensures that there is n arc in the solutionconstraint (12) ensures when a node is visited at stage t, it is left at stage $t+1$, (13) ensures variable x_{ij} is linked to y_{ij}^t, (14) is a binary constraint and (15) forces exit and entrance at node 1 only at stage one and stage n, respectively. Note that variable x_{ij} is not necessary in this formulation, however, it is used for consistency.

Another formulation is the basic (single-commodity) flow-based formulation proposed by Gavish and Graves (1978). They introduced a flow variable y_{ij} which denotes the flow on arc (i,j)

$$\sum_{j=1}^n y_{ij} + \sum_{j=1}^n y_{ij} = -1, \qquad i = 2,\dots,n \quad (16)$$

$$y_{ij} \leq (n-1)x_{ij}, \quad i,j = 1,\dots,n \quad (17)$$

$$y_{ij} \geq 0, \quad i,j = 1,\dots,n \quad (18)$$

where (16) states that the total flow entering any node minus the total flow leaving the same node is equal to its demand (where the demand of one unit represents a visit to the node), (17) states that there is a positive flow on arc (i, j) if and only if it is used by the salesman and (18) states non-negativity requirements. For a survey and comparison of the aforementioned TSP formulations, refer to Orman and Williams (2006).

2.2. Properties and relaxations

TSP has two main properties. The first property is the connectivity of the tour, which means there is always a path between any pair of vertices and the second is the degree of every vertex in the TSP which is two, meaning that for any arc that enters the vertex there should be another arc leaving that vertex. The assignment problem and 1-tree possess one but not both properties of the TSP (Christofides, 1975). The assignment problem (AP) holds the second property but not necessarily the first one. On the other hand, 1-tree holds the first property and not necessarily the second one. Since 1-tree and AP relaxation are much easier to solve than the TSP, they can be used as relaxations of the TSP and the cost of their (typically infeasible) solutions could be used to initialise the dual bound, which in a minimisation context represent the lower bound, and such infeasible solutions could be improved/repaired either by exact solution methods or by heuristic solution methods.

AP has been the first relaxation of TSP, which is obtained by objective function (1) and constraints (2, 3, and 5). Eastman (1958), Little et al. (1963), Shapiro (1966), Bellmore and Malone (1971), Smith et al. (1977), Balas and Christofides (1981), and Miller and Pekny (1991) were among researchers that used AP-relaxation to solve TSP within branch-and-bound methods. On the other hand, 1-tree is a minimum spanning tree (MST) with an extra minimum arc at node one (Christofides, 1975; Held and Karp, 1970, 1971). A 1-tree mathematical formulation proposed by Held and Karp (1970, 1971) minimises (l) under the following constraints:

$$\sum_{i \in V} \sum_{j \in V} x_{ij} = n, \qquad 19$$

$$\sum_{j\in V} x_{1j} + \sum_{j\in V} x_{j1} = 2, \quad 20$$

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1, \ \forall S \subset V' - \{1\}, S \neq \emptyset \quad 21$$

$$x_{ij} = 0 \ or \ 1 \ , \ i,j \in V \quad 22$$

where (19) ensures that 1-tree has n edges, (20) ensures that vertex one has degree two, (21) ensures that no cycle exist in 1-tree except at node 1 and (22) is the binary constraint. Held and Karp (1970, 1971), Christofides (1970), Smith and Thompson (1977), Volgenant and Jonker (1982), Gavish and Srikanth (1983) were among the first to use 1-tree relaxation for solving TSPs.

Other less popular relaxations of the TSP are 2-matching problem relaxation (Bellmore and Malone, 1971) and shortest n-arc path problem relaxation (Houck et al., 1980). In this research, we proposed repair mechanisms for solutions of AP-relaxations of the TSP.

2.3. Exact methods

Exact solution approaches solve COPs to optimality. However, these approaches are in general computationally intensive and their efficiency depends on the choice of the bounding schema used to prune nodes in the search tree to avoid exploring branches with no potential to deliver an optimal solution. Solution methods within this category can be categorised as branch-and-bound algorithms, cutting plane algorithms, and their hybrids as branch-and-cut algorithms.

One of the common methods used to solve TSP is the branch-and-bound (B&B) methodology. B&B is a solution strategy based on the "divide and conquer" principle. The idea is to partition the feasible region of an integer linear programming problem (ILP) into more manageable subdivisions and then to further partition the subdivisions, if necessary. This partitioning process of the solution space is referred to as the branching process. In order to avoid unnecessary branching, a bounding scheme is used. The branching process may be viewed as a successively finer and finer subdivision of the feasible region, where each subset in a given partition represents a subproblem. The branching process may also be viewed as a tree where the root

represents the linear programming (LP) relaxation of the original ILP and each other node represents a subproblem.

Moreover, a bounding scheme is used to eliminate some nodes in the B&B tree in order to reduce computational requirements. The bounding scheme should be designed so that, during the course of the algorithm, a decreasing sequence of upper bounds and an increasing sequence of lower bounds are produced, and the algorithm stops when such sequences converge to the same value. In the case of TSP, the lower bound can be computed by any of its relaxations, such as AP-relaxation. Eastman (1958), Little et al. (1963), Shapiro (1966), Bellmore and Malone (1971), Smith et al. (1977), Toth (1980), Balas and Christofides (1981), Miller and Pekny (1991), and Turkensteen et al. (2006) are some of those researchers that used AP-relaxation to solve TSP with branch-and-bound methods.

In order to reduce the size of the tree one might use cutting plane methods to diminish part of the feasible region. These methods were proposed by Ralph Gomory in 1950s (Gomory, 1958) to solve integer linear programming (ILP) and mixed-integer linear programming problems (MILP). If a linear constraint is added to an integer linear programme (ILP) that does not exclude integer feasible points, called a cutting-plane or cut, then the solution is unchanged. Cutting-planes have the effect of lopping off part of the feasible set, but no integer points are lost. The main idea of cutting plane algorithms is adding cuts to an ILP, one at a time, until the solution to the LP relaxation is integer. Because no integer feasible points have been excluded, the final solution to the relaxed ILP with added constraints will solve the original ILP. Fleischmann (1988), Miliotis (1978) and Avella et al. (2017) are amongst those researchers who used cutting palne methods to solve TSP and other routing problems.

The difference between B&B and cutting plane methods is that, at each stage of the B&B algorithm, the current feasible region is cut into two smaller regions by the imposition of two new constraints, whereas at each stage of the cutting-plane algorithm, the current feasible region is diminished, without being split, by the imposition of a single new constraint. Splitting (respectively lopping off) is done so that the optimal solution to the current program must show up as the optimal solution to one of the two new programs (respectively the new program).

Later, to strengthen these methods and minimise their drawbacks, researchers combined exact methods' strategies and proposed hybrid exact methods. One of these hybrids is a combination of B&B and cutting plane methods within a single framework, called branch and cut (B&C), which uses cuts to reduce the size of the tree (Crowder and Padberg, 1980; Padberg and Rinaldi, 1991; Fischetti and Toth, 1997; Fischetti et al, 2003; Applegate et al, 2007).

2.4. Heuristic methods

In a trade-off between time and optimality, one might prefer near-optimal solutions that use the least possible time. Although heuristic methods cannot prove optimality of the solution, they can find near-optimal solutions quickly. Furthermore, they are relatively easier to explain, implement and adapt to different problems. Heuristic approaches can be categorised into tour construction procedures, tour improvement procedures and composite algorithms (Laporte, 1992). Tour construction heuristics add or insert a vertex to a tour, one at a time. While inserting or adding a node, the tour cannot be improved during the construction procedure. These heuristics can be categorised as nearest neighbour procedure (Rosenkrantz, Steams and Lewis, 1977); insertion procedures (Rosenkrantz, Sterns and Lewis, 1977); Clarke and Wright savings procedures (Clarke and Wright, 1964); minimal spanning tree procedure (Kim, 1975); Christofides heuristic (Christofides, 1976); partitioning procedure (Karp, 1977); nearest merger procedures (Rosenkrantz et al., 1977; Glover et al., 2001); path merging procedures (Yeo, 1997; Glover et al., 2001); contract or patch algorithm (Glover et al., 2001); and GENI (Gendreau, Hertz and Laporte, 1992). For more detail see Appendix A.

Later, one can improve the tour obtained by the tour construction procedure using tour improvement methods. Tour improvement procedures start with an initial feasible solution, obtained either by one of the tour construction methods or randomly, and exploit all its neighbours for a better feasible solution. Local search is one of the tour improvement procedures which improve the tours by one or more than one

neighbourhood moves and a selection strategy. A comparative analysis of tour construction and local search procedures can be found in Golden et al. (1980).

In this thesis, we only made use of 2-opt (flood, 1956; Croes, 1958), 3-opt (Bock, 1958; Lin, 1965) and Or-opt (Or, 1976) neighbourhood structures. k -opt neighbourhood was first introduced by Lin (1965) for the TSP. k -opt involves removing k arcs from the tour and replacing them with k new arcs. Two specific cases of k -opt are 2-opt and 3-opt neighbourhood moves which are the most used neighbourhood structure in the literature. The 2-Opt moves respectively 3-opt, consist of deleting two arcs, respectively three arcs, and reconnecting the resulting paths, see Figure 1 and Figure 2. Or-opt (Or, 1976) is a modified version of 3-opt which considers relocation of a string of 1, 2 or 3 nodes in the tour.

Moreover, one can use partial destruction/construction (D/C) neighbourhood moves to improve the tour. These moves start with a complete solution and iteratively destruct a part(s) of the tour and reconstruct it using construction heuristics. Ruin-and-recreate heuristic (R&R), proposed by Schrimpf et al. (2000), is a more general concept of destruction/construction moves. R&R ruins parts of the solution and reinsert them using an insertion procedure. In comparison with other D/C moves, R&R destruction considers larger areas (e.g. more nodes).

The before mentioned basic local search methods iteratively explore all the search space for a better solution and stop when the search cannot find a better solution. A better solution is selected by an acceptance strategy which can either be first improvement or best improvement. The first improvement acceptance strategy stops the search at each of the iterations as soon as a better solution is found. On the other hand, the best improvement acceptance strategy at each of the iterations explores all neighbourhoods and returns the best solution. Both acceptance strategies find the local optimum. A comparative analysis of these two acceptance strategies can be found in Hansen and Mladenović (2006).

The goal of using construction heuristics and tour improvement procedures is finding a good quality tour. To do so, these procedures search within all possible edges. However, most of the possible edges do not exist in the optimal tour, especially the

long ones. In other words, discarding these edges from the search space will speed up the search and might improve the heuristic's performance. Reinelt (1994) proposed

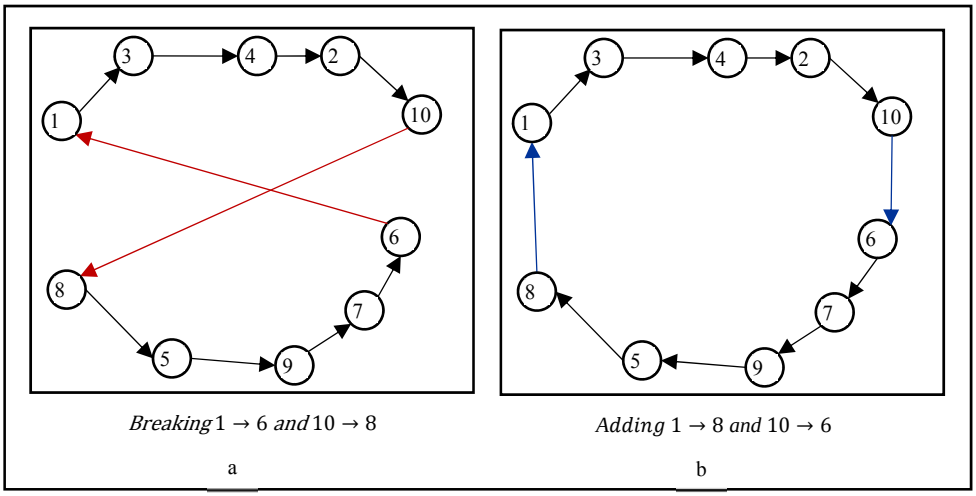


Figure 1 2-opt move

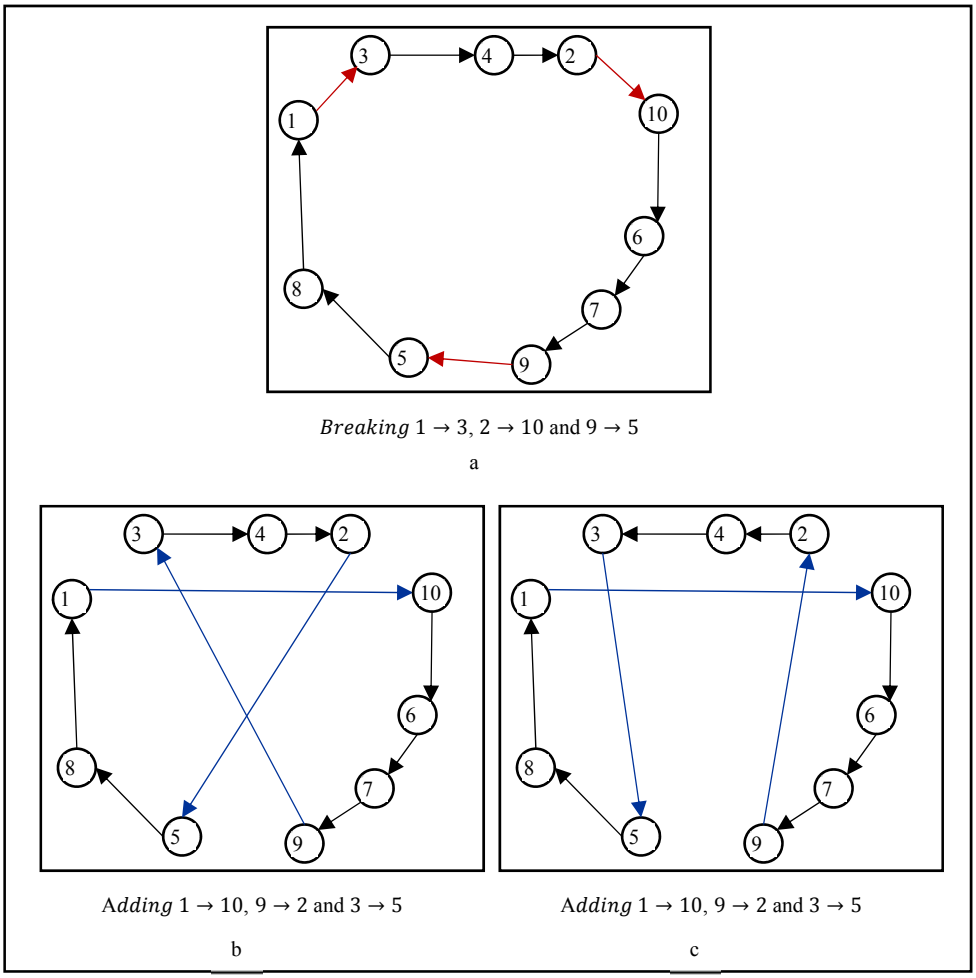


Figure 2 3-opt move

variations of these heuristics, by making use of candidate sets in heuristic procedures, to enhance their performance. A candidate set, say CS, of an optimisation problem, say P, is a subset of the feasible space of P defined to narrow down the search space to "promising" regions as specified by a set of criteria that exploit the domain knowledge of the problem. Using candidate set has advantages and disadvantages. The main advantage of a candidate set is speeding up the search process. On the other hand, the use of candidate sets could have disadvantages (e.g., the choice of the criteria for defining a candidate set could result in discarding an optimal solution or the path to an optimal solution) if not chosen appropriately.

The simplest and most well-known candidate set is K-Nearest Neighbour (K-NN). K-NN was first introduced by Fix and Hodges (1951) in an unpublished US Air Force School of Aviation Medicine report. The K-NN candidate set limits the search for the nearest neighbour of i to the subgraph of G that consists of the K nearest neighbours only, say $G_K = (N, A_K)$, where

$$A_K = \{(i,j) \in A | \; j \; is \; amongst \; the \; K \; nearest \; neighours \; of \; i\}.$$

However, G_K might itself consist of disconnected subgraphs or clusters, depending on the structure of the original graph G (e.g. geometrical properties). Depending on the choice of the value of the parameter K, K-NN have consequences, for example, searching in the K-NN subgraph might force the method to stop without delivering a complete feasible solution, if K is small enough to result in disconnected subgraphs or clusters. On the other hand, searching in the K-NN subgraph is more likely to deliver a better-quality solution than the one delivered by the original construction heuristic or local search; for example, searching in the K-NN subgraph might deliver better quality solution than the original nearest neighbour heuristic, as it avoids the drawback of a greedy search that ends up adding long arcs near the end of the construction process (Reinelt, 1994). Reinelt (1994) proposed using candidate set-based variant of construction heuristics and tour improvement heuristics. For example, Candidate set-based variant of insertion heuristics (Reinelt, 1994) expands the current subtour S by inserting a node from the candidate set of nodes in the subtour based on prespecified criteria, such as arbitrary insertion, nearest insertion, cheapest insertion, and farthest insertion. In candidate-based variant of improvement procedures (Reinelt, 1994), the

only moves considered are the ones where at least one of the edges is already in the candidate set.

Another drawback of local search method is that the minimum solution found might not be a good quality solution. In other words, the obtained local minimum solution can be far from the optimal solution. However, an extension of local search, called metaheuristics are designed to escape the local optima and look for a better solution.

2.5. Metaheuristics

A metaheuristic provides guidance mechanisms or strategies for searching the solution space effectively and often avoid remaining stuck in local optima when encountered (Sörensen and Glover, 2013). An effective and successful metaheuristic search strategy balances exploitation (intensification) of the search around the best solution found so far and exploration (diversification) of the search space. The most common classification of metaheuristics is sequential (single-point or trajectory) metaheuristics and parallel (population-based or evolutionary) metaheuristics. The first category, sequential metaheuristics, are more exploration-based whereas the second category, parallel metaheuristic, are more exploration based. Sequential metaheuristic start with a single solution and attempt to improve it by searching its neighbourhood. In other words, they exploit the promising search areas that might lead to a good quality solution. On the other hand, parallel metaheuristics such as genetic algorithms start the search with a set of solutions and attempt to find better solutions by iteratively combining them in the hope that they keep the best features of the older solutions, while diversifying the search towards new areas that has not been explored before. In the next subsections an overview of these metaheuristics is presented.

2.5.1. Sequential metaheuristics

As it was mentioned earlier, sequential (single-point) metaheuristics, such as, simulated annealing (Kirkpatrick et al., 1983), tabu search (Fred Glover, 1986, 1989, 1990) and variable neighbourhood search (Mladenović and Hansen, 1997), start with a single solution, using a sequential search strategy, search its neighbourhood for a

better solution. The implementation of these metaheuristics involves several decisions to be made, which we classified them into problem-specific decisions and generic decisions. Problem-specific decisions are related to the nature of the problem and require a deep knowledge of the problem. In contrast, generic decisions can be taken without such knowledge since they are specific to the chosen metaheuristic and not the problem. First, we present problem-specific decisions since they are common for all sequential metaheuristics. Then, the aforementioned sequential metaheuristics and their generic decisions are explained in more details.

I. Problem-specific decisions

The decisions common to the implementation of SA, TS and VNS are (1) choice of the solution space; (2) choice of the form of the objective function; (3) choice of initial solution; and (4) choice of the neighbourhood structure or type of moves to use. These decisions are similar across the single-point metaheuristics.

(1) Choice of the solution space

In principle, all possible solutions are admissible. However, for computational reasons, one might want to reduce the size of the solution space to converge to a good solution faster. This strategy might limit the search to a small neighbourhood, which might not include the global optima. On the other hand, one might permit infeasible solutions (by permitting constraint violations) to increase the search space and the possibilities of finding the global optima (Gendreau et al., 1994; Glover, 1977).

(2) Choice of the form of the objective function

The value of the objective function is used to discriminate between solutions and to decide whether the search should move to a new neighbour or not. The original objective function is the objective function of the optimisation problem under consideration; e.g. total distance or cost of a feasible solution to the TSP. In addition to the original objective function, one might define other types of objective functions, such as the surrogate objective function, the auxiliary objective function, and the penalised objective function.

The surrogate objective function **(Crainic et al., 1993):** Calculating this objective function is much faster than the original one since instead of calculating the total cost

of the solution, only the cost incurred by the move is calculated, which is correlated to the original objective function.

The auxiliary objective function **(Gendreau, 1993):** This objective function measures the desired attributes of the solution, instead of calculating the original objective function. This objective function is used when the original objective function does not provide knowledge and information about the search space. For example, in routing problems, when searching the neighbourhood of the current solution faces a tie (several new solutions with equal cost), one can introduce an auxiliary function including more desirable attributes (such as travel time, customer's demand, etc.) to direct the search. Auxiliary objective functions are also used when one allows for infeasibilities but penalises them.

The penalised objective function **(Gendreau et al., 1994; Glover, 1977)**: This objective function adds penalty terms to penalise violations of constraints or features of solutions, this is done by relaxing some of the constraints and adding a penalty, for each violation, to the objective function. However, one should choose the correct penalty. This strategy (i.e. allowing infeasibilities) widens the search space.

(3) Choice of the initial solution

The initial solution can be generated by either a random procedure or a simple heuristic. Starting with a random solution might not lead to the optimal solution or a slow convergence to a good solution. On the other hand, starting with a good solution, e.g. obtained by a construction heuristic, might lead to quick convergence, although the search might get stuck in a local optimum.

(4) Choice of the neighbourhood structure or type of moves to use

In designing metaheuristics, neighbourhoods are defined to move from a solution in the search space to another. Designing neighbourhood structures depends on the characteristics of the solution space and the type of moves used to move from a solution to its neighbour. For example, to solve TSP using metaheuristic, one can use either one or all the following moves: k-opt, or-opt, node exchange, node relocation, string exchange, etc. (flood, 1956; Croes, 1958; Bock, 1958; Lin, 1965; Or, 1976). A small neighbourhood structure is preferable since it typically converges toward a good quality solution much quicker, however, it might not lead to a good quality solution.

For instance, a 2-opt move (e.g. k-opt when $k = 2$) is much faster than a 3-opt move (e.g. k-opt when $k = 3$), however, 3-opt produces better quality solutions than 2-opt.

II. Simulated annealing and its generic decisions

Simulated annealing (SA) is a search procedure based on the annealing process of materials in metallurgy and the underlying thermodynamic laws introduced in the early 1980s. The initial design of SA has been proposed by Kirkpatrick et al. (1983). Its main search strategy consists of avoiding remaining stuck in a local optimum by temporarily accepting worse solutions with some probability, where this probability decreases as the search progresses.

Initialisation Step

Choose a heuristic to initiate the initial solution, say x_0, and set $Z(x_0)$ to the total distance of the TSP tour x_0;

Initialise the best solution found so far, by setting $x^* = x_0$ and $Z^* = Z(x_0)$;

Choose an initial temperature $\tau^0 > 0$ and set the current temperature $\tau = \tau^0$;

Set the temperature change counter $t = 1$;

Iterative Step

REPEAT until stopping condition = true

  Choose the number of neighbours to visit at the current temperature τ, r_τ;

  Set the repetition counter $r = 0$;

  REPEAT until stopping condition = true // e.g., $r = r_{max}$

    Generate randomly a neighbour x of the current seed x_0 and compute $Z(x)$;

    Compute the change δ in the objective function value: $\delta = Z(x_0) - Z(x)$;

    IF $\delta > 0$ **OR** $Random(0,1) < acceptance\ probability$ **THEN {**

      Update the current seed solution (Z_0, x_0); that is, set $x_0 = x$, and $Z_0 = Z$;

      IF $Z^* > Z(x)$ **THEN**

        Update the best solution found so far (Z^*, x^*); that is, set $x^* = x$ and $Z^* = Z(x)$;

    }

    Increment the repetition counter by 1; that is, set $r = r + 1$;

  END REPEAT;

  Increment the temperature change counter by 1; that is, set $t = t + 1$;

  Reduce the temperature τ according to the temperature reduction function α; that is, set $\tau = \alpha(\tau)$;

END REPEAT

Table 1 Pseudo-code of SA

The basic SA starts the search with a random (initial) temperature,τ_0, increases the temperature until a prespecified threshold, then cools slowly until it reaches a frozen state or final temperature, τ_f. This process requires a cooling schedule that will be explained in the next section. In each SA state or epoch, where each state of SA is distinguished by temperature τ_t, several neighbours of the current solution are visited, and the current solution is updated considering a transition mechanism. The pseudo-code the SA algorithm is outlined in Table 1.

Since it was first introduced, several implementations of SA have been proposed (Lundy, 1985; Johnson et al., 1989, Connolly, 1990, 1992; Cordeau et al., 1997,2001; Koulamas et al., 1994; Ho and Haugland, 2004; Geng et al., 2011; etc.). In general, an implementation of SA requires a number of generic decisions to be made, namely an annealing schedule and an acceptance function (AF). Hereafter, we shall discuss these implementation decisions in more detail.

(1) The annealing schedule

The annealing schedule controls the temperature in SA algorithm, which involves several parameter choices:

- Initial and final temperature
- Number of neighbours to visit at each temperature
- Temperature change strategy and the form of the temperature change function
- Stopping criterion or freezing state

These parameters could be either static or dynamic. When a parameter is constant or fixed throughout the algorithm, from the start to the end, it is a static parameter, and if it is not constant or fixed, meaning that the parameter is changing to adapt to the algorithm, it is a dynamic parameter (Aarts et al., 2014). In implementing the optimal annealing schedule for SA algorithm, one should specify the best choice for these parameters. Many annealing schedules have been proposed which will be summarised in the next section.

Initial and final temperature

The initial temperature, τ_0, should be high enough so that all new neighbours are accepted (Kirkpatrick et al., 1983). Kirkpatrick et al. (1983) proposed starting the

search with a random initial temperature and heating the system by doubling the temperature until the percentage transitions in that epoch are less than P_0. When this percentage is achieved, cooling the system starts. The cooling process continues until the temperature reaches the final temperature, τ_f, which it is zero or close to zero. When the temperature is zero, no uphill moves will be accepted.

Lundy (1985) proposed setting τ_0 to the upper bound of the highest objective function value or proportional to $\sqrt{(n-1)}$, $\tau_0 = 10\sqrt{(n-1)}$ and setting the final temperature to $t_f = \frac{1}{nlog\,n}$. Later, Lundy and Mees (1986) used the same idea to initialise τ_0; however, they proposed setting the upper bound for TSP to the sum of longest edge leaving each city. They also proposed freezing the system when $t < {}^{-1}/_{\log P_{min}}$, where P_{min} is a predefined small probability.

On the other hand, Johnson et al. (1989) considered $\tau_0 = \bar{\delta}/ln(P_0)$, where $\bar{\delta}$ denotes the average increase in the objective function values, computed with uphill moves only, obtained during prespecified number of trials/transitions of the annealing process with the fraction of accepted uphill transitions equal to P_0. Later, Connolly (1990, 1992) proposed an approach to avoid choosing an initial value of P_0. They proposed setting the initial temperature $\tau_0 = \delta_{min} + (\delta_{max} - \delta_{min})/10$, where δ_{min} and δ_{max} denoting the minimum value and the maximum value of the objective function over a number of trial runs, for a range of fixed temperature; and setting the freezing temperature to the maximum value of the objective function over the trial runs, δ_{max}.

Parthasarathy and Rajendran (1997b) defined δ as the relative percentage change in the objective function value; i.e., $\delta = \frac{(z(x_0)-z(x))\times 100}{z(x_0)}$. In addition, they chose to accept solutions with lower quality by 50% relative to the initial solution, with an acceptance probability P_0 equal to 90%. Implicitly, they assume that the maximum relative percentage difference in cost between neighbours is 50% and compute τ^0 accordingly:

$$P_0 = e^{\delta/\tau^0} \Leftrightarrow 0.9 = e^{-50/\tau^0} \Leftrightarrow \tau^0 = 475. \qquad 23$$

Correspondingly, they fixed the final temperature to a value computed based on the initial temperature, prespecified cooling ratio, say α, and the prespecified number of epochs, say t_{max}.

$$t_f = \tau^0 \times \alpha^{t_{max}} \quad 24$$

The number of neighbours to visit at each temperature

The number of neighbours to visit, r_t, at each epoch is typically set to a fixed prespecified value, which may depend on the solution space or neighbourhoods (Ogbu, 1990) However, the best choice for this value should consider the temperature change strategy, the form of the temperature change function and the value(s) of its parameter(s) and the choice of the stopping criteria. Moreover, one might use feedback from the annealing process, such as the ratio of acceptance or the minimum number of accepted moves to choose the value of r_t.

The temperature change strategy

The performance of SA algorithm is highly dependent on the temperature change strategy. A fast cooling strategy might speed up the process, but it might also lead to local optima far from the global one. On the other hand, a slow cooling strategy might lead to an optimal or near optimal solution in a very long time. Thus, when designing an optimal cooling schedule, one should make a trade-off between the quality and CPU time.

In addition, a temperature change strategy could be constant (fixed temperature), non-adaptive or adaptive. Non-adaptive temperature change strategies are systematically decrease based on a cooling function. By contrast, adaptive strategies decrease, based on a cooling function, and increase the temperature, based on a heating function, as required and based on prespecified conditions. In this section an overview of these temperature change strategies is presented.

The first proposed temperature change strategy consists of two steps: first melting and then cooling the system. Kirkpatrick et al. (1983, 1984) proposed starting the schedule with a random initial temperature and heating the system, by doubling the temperature, until the percentage accepted moves reach the prespecified threshold. Then, the

cooling process starts using a temperature reduction function $\alpha(t)$. This schedule is called the Geometric Schedule (18), which is one of the most popular cooling strategies

$$\alpha(t) = \alpha.\tau \Leftrightarrow \tau_t = \alpha.\tau_{t-1}; 0 \prec \alpha \prec 1 \quad 25$$

where α is the cooling ratio. If the total number of epochs, t_{max}, is prespecified as well as τ_0 and τ_f, then the cooling ratio α can be computed as follows:

$$\alpha = \left(\frac{\tau_f}{\tau_0}\right)^{1/t_{max}}. \quad 26$$

Typically, most of the authors are making use of this cooling strategy by fixing α ($\alpha \prec 1$) to values in range between 0.8 and 0.99. For more detail on other popular cooling strategies see Appendix B. As for comparative analysis among different cooling strategies refer to Mirkin et al. (1993), Steinhöfel et al (1998) and Nourani and Anderson (1998). Park and Kim (1998) proposed a systematic procedure to choose the appropriate values for the parameters of SA. Their procedure chooses the parameter values of the cooling schedule.

(2) Stopping criteria

The basic SA stops when the system freezes that is when the temperature reaches freezing point or final temperature, which is normally equal to zero (Kirkpatrick et al., 1983). However, SA might take longer to stop. Later, several stopping criteria other than the 'freezing' state of the system were proposed (Salhi, 2017), such as the number of iterations or temperatures or epochs reaches a prespecified number, computational time exceeds a prespecified time limit, the maximum number of temperature changes without improvement of the current seed is reached, the best objective function value found so far is not updated for a prespecified number of iterations, etc.

(3)* Acceptance *Function

Since SA is not a greedy algorithm, it accepts a neighbour of the current solution as a new seed if it is either an improving one or a non-improving one but satisfies a second criterion called the acceptance criterion.

Kirkpatrick et al. (1983) used Metropolis criterion to accept new neighbours with a probability. This acceptance probability function (APF), which is dependent on the quality of the new neighbour and the current temperature of the system, is as follows:

$$APF(\delta, \tau_t) = \begin{cases} exp\left\{-\frac{\delta}{\tau_t}\right\} & \delta < 0 \\ 1 & \delta \geq 0 \end{cases} \quad 27$$

where δ indicates the energy change of the move and is calculated as $\delta = Z(x_0) - Z(x)$. Similarly, Connolly (1990) proposed the following acceptance function (AF) where k is the Boltzmann's constant.

$$APF(\delta, t_i) = \begin{cases} exp\left\{-\frac{\delta}{k\tau_t}\right\} & \delta < 0 \\ 1 & \delta \geq 0 \end{cases} \quad 28$$

When k is equal to one, AF would be equivalent to Kirkpatrick et al. (1983) proposed APF.

On the other hand, Johnson et al. (1989) proposed a linear AF, equation (22), instead of the exponential function. This function is faster, and as they mentioned in their paper, it has a significant difference in quality with the exponential function.

$$APF(\delta, \tau) = 1 - \frac{\delta}{\tau} \quad 29$$

For more detail, on other popular acceptance functions, see Appendix C.

III. Tabu search and its generic decisions

Tabu search (TS) is a memory-based metaheuristic, introduced by Fred Glover (1986, 1989, 1990), which explores the neighbourhood search space strategically and guides the local search out of local optima and towards global optimality. In other words, making use of an adaptive memory and responsive exploration in TS (Glover and Laguna, 1997) could lead the search to a new neighbourhood by reducing the likelihood of cycling or remaining stuck in a local optimum. The Pseudo-code of the basic TS presented in Table 2, is based on best improvement local search and a short-term memory. Several implementations have been proposed for the TS metaheuristic (Glover, 1986, 1989, 1990; Hertz and de Werra, 1987; Glover and Laguna, 1997; Taillard, 1990, 1991; Cordeau et al., 1997, 2001; He et al., 2005; Archetti et al., 2006; etc.). However, implementation of TS metaheuristic requires several generic decisions to be made; namely tabu moves, memory, search strategies, transition mechanism, aspiration criteria and stopping criteria. Hereafter, we shall discuss these implementation decisions in more detail.

Initialisation Step

Choose a heuristic to initiate the initial solution, say x_0, and set $Z(x_0)$ to the total distance of the TSP tour x_0;

Specify the aspiration level function and initialise its value;

Choose the tabu list (TL) size and initialise TL to the empty set $\emptyset$;

Set iteration counter I to 0;

Iterative Step

REPEAT until stopping condition = true

Find a neighbour, say x, of the current TSP tour x_0

IF x is not tabu **THEN**

Update the current seed solution (Z_0, x_0); that is, set $x_0 = x$, and $Z_0 = Z$;

ELSE

IF x is tabu but the aspiration criterion overrides its tabu status; e.g., x is better than the best neighbour found so far **THEN**

Update the current seed solution (Z_0, x_0);

ELSE

Find the best non-tabu neighbour x– rather than an improving one – in the neighbourhood of the current neighbour x_0 and update the current seed solution (Z_0, x_0);

Update the tabu list TL;

IF $Z(x) < Z^*$ **THEN** update the best solution found so far (Z^*, x^*); i.e. set $x^* = x$ and $z^* = z(x)$;

Increment iteration counter by 1; that is, set $I = I + 1$;

END REPEAT

Table 2 Pseudo-code of TS

(1) Tabu moves

Tabu moves are defined as forbidden moves, to prevent cycling (tabu restrictions). Some advantages of stating some moves as tabu are to avoid being stuck in local optima and to widen the exploration space by forcing the search to explore new neighbourhood areas. Although these new neighbourhood areas might not include the global optimum, they may lead the search to it. On the other hand, these areas might lead the search far away from the global optimum.

Commonly used tabus involve keeping track of the most recent moves leading to the current solution and preventing the reversal of these moves to stop cycling back to previous local optima or solutions, while other tabus only keep key characteristics of solutions or moves (Gendreau and Potvin, 2014).

The most regularly used tabu list is a circular list with fixed length (Glover 1986; Hertz & de Werra 1987) which forbids cycling back to the most recent moves for several iterations. Tabu tenure is the number of iterations a move is forbidden. In standard TS, tabu tenure is fixed, although one might define a dynamic procedure to change the tabu tenure throughout TS (Glover 1989, 1990; Skorin-Kapov 1990; Taillard 1990, 1991).

(2) Memory

The use of memory in TS is for keeping the search history to guide the neighbourhood search. The general TS framework makes use of three types of memories; commonly referred to as short-term memory, intermediate-term memory, and long-term memory. Each type of memory could be used in a different configuration of the neighbourhood search. For example, one might be used to restrict, while the other might be used to widen the neighbourhood search.

A tabu search with the first type of memory, short-term memory component, is a constrained greedy search process that seeks to make the best move to satisfy certain constraints embedded in the tabu restrictions designed to prevent cycling. These tabu restrictions do not operate in an isolated manner but are counterbalanced by the application of aspiration criterion. The intermediate-term memory component is used in an intensification process that drives the search into regions with features that were historically, during the search process, found to be good. Finally, the third and last type, the long-term memory component, is used in a diversification process that drives the search into new regions that contrast with those examined so far.

For computational reasons, most TS implementations make use of the short-term memory component only. The core of TS is embedded in its short-term memory component, and many of the strategic considerations underlying this process reappear, amplified in degree but not greatly changed in kind, in the intermediate-term memory component (intensification process) and the long-term memory component (diversification process).

Furthermore, memory could be either explicit or attributive. Explicit memories record the full solutions, typically local optimums, whereas attributive memories, the most commonly used, only keep track of the key characteristics of the changes that lead to current solutions. The memory used in TS records either the most recent or the most

frequent solutions or attributes. The first memory structure is called recency-based memory, which keeps track of the most recent solutions (Glover 1986, 1989, 1990; Hertz and de Werra 1987; Friden et al. 1989; Skorin-Kapov 1990; Taillard 1990, 1991; Montané and Galvão 2006; Ho and Haugland 2004; Archetti et al. 2006). The second is called frequency-based memory that tracks the moves and the number of iterations they occurred.

These memory structures could be integrated, for example, a recency-based short-term memory could be combined with frequency-based long-term (or intermediate-term) memory both /either to diversify and/or to intensify the search (Cordeau et al. 1997, 2001; Montané and Galvão, 2006).

(3) Search strategies

The most important part of the tabu search is its search strategy. Tabu search uses intensification and diversification strategies to guide the search away from the local optima and towards the global optimum. The key to implementing a good TS is in balancing these two strategies.

Intensification strategy

Intensification strategies search the promising neighbourhood areas more thoroughly. These promising neighbourhood areas are those of the local optima. Intensification strategies make use of intermediate-term memory, such as a recency-based memory structure recording the complete local optimums or recording the number of consecutive iterations where various solution elements were present in the local optimums.

A typical intensification approach is restating the search from the best-known solution by fixing several attractive components using intermediate-term recency-based memory. Another approach is restating the search from the best-known solution and intensifying the search for a better solution by applying a simple local search with a different neighbourhood structure for several iterations (Renaud et al., 1996; Ho and Haugland, 2004). A third approach is a continuous intensification that uses a type of intermediate or long-term frequency-based memory. In this approach, it is more

attractive to insert components with a greater weight related to their frequencies (Montané & Galvão, 2006).

Diversification strategy

Diversification strategies force the search into unexplored regions of the search space. Although using short-term memory in basic TS enforces diversification to some extent, it could be based on other types of long-term memory, such as frequency-based memory recording the number of iterations (from the beginning) where various solution elements have been present in the current solution.

A simple way to diversify the search is by restarting the search with a new solution generated randomly or by a heuristic. Another approach is to use long-term frequency-based memory. In this approach, the search starts from a new solution obtained by introducing a number of elements with the lowest frequency. An alternative frequency-based approach is called continuous diversification, where a frequency-based penalty is added to the objective function (Cordeau et al. 1997, 2001). This approach penalises the most frequent elements. Another continuous diversification is inserting components with lower frequency based on some weight related to their frequencies (Montané & Galvão, 2006).

Most of the implemented TS methods only search in primal search space, i.e. only feasible solutions are allowed. Since primal search space limits the possibilities, it can lead to a local optimum and not the global optimum. One way of overcoming this problem is to allow infeasibilities by constraint relaxation, which will widen the search space. Constraint relaxation removes a constraint from the problem and adds penalty terms for each constraint violation to the objective function (Gendreau et al. 1994; Cordeau et al. 1997, 2001).

(4) Aspiration criteria

As previously mentioned, tabus are used to prevent cycling back to previous moves since they may lead to a better solution. Aspiration criterion (AC) defines a mechanism to cancel tabu status if the criterion is satisfied. The simplest and most common form of the aspiration criteria is that if the solution found by a tabu move is better than the best-known solution throughout the search, the tabu status of that move will be

cancelled and the move will be accepted since, obviously, this new solution has not been visited yet. In TS methods that allow infeasibility, the aspiration criteria could be allowing a better feasible solution than the current best-known solution.

(5) Transition mechanism

The transition mechanism in TS can be best described as a constrained steepest descent, where the adjective "constrained" refers to the tabu restrictions. In other words, a transition is accepted only with consideration of the tabu restrictions and the acceptance criterion.

(6) Stopping criteria

Theoretically, the search continues until the global optimum is found. Since the best solution is not known or assumed to be unknown, one should decide on when the search stops. Several stopping criteria can be used such as when the maximum number of iterations is reached, the maximum number of iterations without improvement of the current seed is reached, the computational time exceeds a pre-specified time limit, the best objective function value found so far is not updated for a pre-specified number of iterations, the objective function value reaches a threshold, etc.

As for the comparative analysis between TS and other metaheuristics, several comparative analysis has been done, however their comparative analysis is uncertain, to some extent. Sinclair (1993), Paulli (1993), Battiti and Tecchiolli (1994), Chiang and Chiang (1998), Arostegui et al. (2006) and Hussin and Stützle (2014) made such comparison between SA and TS for QAP, their results implies that TS outperforms SA. On the other hand, Paulli (1993) implied that when considering the same computational time SA performs better than TS. Later, Hussin and Stützle (2014) suggested that the performance of TS and SA, and whether one is better than the other is dependent on the problem size.

IV. Variable neighbourhood search and its generic decisions

Variable neighbourhood search (VNS) algorithms, proposed by Mladenović and Hansen (1997), is an extension of classical local search algorithms where attempts are made to avoid being trapped in a local optimum by systematically changing neighbourhood structures during a local search process. The Pseudo-code of VNS is shown in Table 3.

Initialisation Step

Choose a heuristic to initiate the initial solution, say x_0, and set $Z(x_0)$ to the total distance of the TSP tour x_0;

Initialise the best solution found so far, say (Z^*, x^*), by setting $x^* = x_0$ and $Z^* = z(Z_0)$;

Choose a set of neighbourhood structures to use and specify the order according to which they will be used, say $\{N_s; s = 1, \dots, s_{max}\}$;

Choose the local search method to use in exploring neighbourhoods;

Initialise neighbourhood structure counter s to 1;

Iterative Step

REPEAT until stopping condition = true

Randomly generate a neighbour, say x, of the current neighbour x_0 according to the s-th neighbourhood structure;

Explore the s-th neighbourhood of x using the chosen local search method and update x accordingly;

IF this local optimum concerning the s-th neighbourhood x is better than the current seed x_0 **THEN**

Update the current seed solution (Z_0, x_0); that is, set $x_0 = x$, and $Z_0 = Z$;

IF $Z(x) < Z^*$ **THEN** {

update the best solution found so far (Z^*, x^*);

Reset neighbourhood structure counter s to 1;

}

ELSE Increment neighbourhood structure counter s by 1;

END REPEAT

Table 3 Pseudo-code of VNS

VNS is based on three facts (Hansen et al., 2016):

- A local optimum obtained from a one neighbourhood structure could not necessarily be obtained by another neighbourhood structure.
- The local optimum of one neighbourhood structure is not necessarily the global optimum, but the global optimum is the local optimum of all neighbourhood structures,
- For many problems, local optima of several neighbourhood structures are close to each other.

Several VNS metaheuristics have been proposed (Mladenović and Hansen, 1997; Burke et al., 2001; Hansen and Mladenović, 1999, 2003; etc.). However, they all

require generic decisions to be made such as the choice of transition mechanism, shaking procedures and stopping criteria, which will be explained in more detail in this section.

(1) Transition mechanism

The transition mechanism is specified through the choices of answers to questions such as: How is a specific neighbourhood of the current seed solution searched? What criteria are used for updating the current seed solution? What criteria are used for changing neighbourhoods? In which order are the neighbourhoods searched?

We shall answer these questions hereafter.

- How is a specific neighbourhood of the current seed solution searched?

Improvement procedures used in VNS could be either random or by using any local search-based procedure or metaheuristic, such as local search, simulated annealing, tabu search, etc. Additionally, one might search the whole neighbourhood or a proportion of it.

- What criteria are used for updating the current seed solution?
  In the neighbourhood exploration step, either the first improvement or the best improvement search strategy could be used. The first improvement strategy accepts the first move causing an improvement, while the best improvement strategy accepts the move with the best improvement among all improving solutions. Additionally, when updating the solution, one might allow solution deterioration, meaning that uphill moves might be accepted with a ratio or probability.
- What criteria are used for changing neighbourhoods?
  Several criteria can be used to change the neighbourhood structures. One might change the neighbourhood structure whenever an improvement occurred or change the neighbourhood structure regardless of the occurrence of improvement. Hansen et al. (2016) classified neighbourhood change strategies as sequential neighbourhood change strategy, cyclic neighbourhood change strategy, pipe neighbourhood change strategy and skewed neighbourhood change strategy; see Appendix D for more detail.

(2) In which order to search the neighbourhoods?

The order in which the neighbourhood structures are searched can be random or according to a specific order, in which case the ordering criteria should be specified. The specific order of changing the neighbourhood structure could be chosen based on the designer knowledge or by using a trial run. In addition, it could be static or dynamic (and may be changed by using a learning mechanism or not).

***(3)* Shaking procedure**

Shaking procedure is used to lead the search out of a trap. Typical and simple shaking procedure is random perturbation of the current solution considering the k^{th} neighbourhood structure. One might consider either diversifying the search by random jump from the current solution or intensifying the search by a small change in the current solution.

(4) Stopping criteria

The typical stopping Criteria for VNS is stopping the search when no further improvement is possible by all the neighbourhood structures, the maximum CPU time or the maximum number of iterations without improvement.

As for comparative analysis between VNS and other local search methods and metaheuristics, a comparison between two variants of VNS and LS for TSP made by Burke et al. (2001) implied that VNS outperforms LS in most problem instances. Later, Hansen and Mladenovic (2003) compared basic VNS, GA and two variants of ant colony methods on scheduling problem. Their results indicated that VNS outperforms others.

2.5.2. Parallel metaheuristics

Parallel (population-based) metaheuristics, such as GAs, start the solution with an initial population and attempt to find a better population by iteratively evolving them. Genetic algorithm was first introduced by John Holland in the early 1970s (Holland, 1975). GA is inspired from the biological process of natural selection and genetic inheritance that preserves a population of individuals or chromosomes and evolves the population using bio-inspired operators, searching for better or best individuals

(Goldberg, 1989; Holland, 1989; Holland, 1975). Some of these bio-inspired operators are selection, evaluation, reproduction and replacement operators.

Initialisation Step

Choose an initial population of M individuals/tours, in the admissible parameter space S evaluate the fitness of each individual, $z(x_m)$;

Initialise the best solution, say (x^*, z^*), among the initial population

Set iteration counter I to 0;

Set *Best-Found-At-iteration* to 0;

Set immigration counter I_{imm} to 0;

Iterative Step

REPEAT until stopping condition = true

IF crossover condition(s) hold **THEN** {

Select a subset of individuals from the current generation as parents for reproduction;

Perform a crossover operation on parents to generate children;

}

IF mutation condition(s) hold **THEN** {

Select a subset of individuals from the current generation as parents for reproduction;

Perform a mutation operation on parents to generate children;

}

IF immigration condition(s) hold **THEN {**

perform an immigration operation to generate children;

Increment immigration counter by 1; that is, set $I_{imm} = I_{imm} + 1$;

}

Evaluate the fitness of each child and update the best solution found so far, if necessary;

IF $z(x') < z^*$ **THEN {**

update the best vector of parameters found so far; that is, set $x^* = x'$ and $Z^* = Z(x')$;

Best-Found-At-iteration $= I$;

}

Replace a subset of parents in the current population by a subset of the current children to produce a new generation;

Increment iteration counter by 1; that is, set $I = I + 1$;

END REPEAT

Table 4 Pseudo-code of GA

The main elements of GA are as follows:

- A **population** consists a set of chromosomes or individuals.

- The **fitness function** is a measure to evaluate the quality of each chromosome. **Genetic operators** are bio-inspired operators, namely selection, evaluation, reproduction, replacement operators.
- **Termination criterion** is used to stop the reproduction process.

A basic GA starts with an initial population and through an iterative process modifies the current population using bio-inspired operators to create a new population, called generation, with a purpose of improving the overall average quality, see Table 4. In each iteration, GA selects a subset of the current population, called parents, to reproduce new individuals, called children or offspring. These new individuals replace a subset of the current population to create a new generation, which is used in the next iteration. This iterative reproduction continues until a stopping condition is satisfied that normally happens when the population converges. Designating a GA requires making two sets of decisions; namely problem-specific decisions and generic decisions. These decisions will be explained hereafter.

I. Problem-specific decisions for GA

The genetic algorithm consists of a population of individuals or chromosomes. Thus, the first step in constructing a GA is to define the genetic representation, also called an encoding scheme, to map feasible solutions of an optimisation problem to chromosomes or strings.

Each chromosome's lifecycle in the population has three phases, namely, birth, life and death. Transformation of each chromosome into each phase and its survival throughout the iterative process mainly depends on its performance or fitness value. After a chromosome's birth, it might be chosen for mating and breeding based on a probability, which is mostly based on its fitness value but not necessarily. The higher the fitness value, the probability of being chosen for mating will be higher. After breeding, the offspring should replace a chromosome in the population, meaning that a chromosome has reached its last phase of existence, death. The most commonly used criterion to choose the chromosome to end its life cycle is choosing the one with the lowest fitness value. Considering the aforementioned decisions, these two decisions, namely the genetic representation of chromosomes and the fitness measure, are problem-specific decisions for GA which will be described in the next section.

(1) Choice of the genetic representation or encoding scheme of chromosomes

A chromosome is a string of genes that keeps the genetic information. Each gene has its position in the chromosome and can have any value from a specific set of alternatives, called alleles. The common representation of chromosomes is binary encoding. When a binary alphabet is not a natural coding for a problem, one may consider an alternative coding with a variety of data structures. Choosing an appropriate encoding is dependent on the type of problem under consideration (McCall, 2005). It is crucial to use an appropriate encoding scheme that adequately describes problem-specific characteristics since it significantly affects all the subsequent steps in the GA such as the form of the reproduction mechanism.

Most of the proposed GAs use fixed length chromosomes for easier implementation of GA operators, although using variable chromosomes can be a better representation for some problems.

(2) Choice of the fitness measure

To imitate the natural law of survival of the fittest, a fitness function needs to be specified to discriminate between chromosomes based on their performance. A variety of fitness measures can be used to evaluate the chromosomes performance. One may use the value of the objective function associated with each chromosome, which might be considered a naïve fitness measure.

Choosing a naïve fitness measure may lead the GA to either converge toward a poor performing chromosome or have a hard time converging toward any solution. A scenario for the first situation, converging to a poor performing chromosome, could occur at the start of the process when most of the chromosomes are weak, and only a few of them are outstanding. In this case, a naïve fitness measure will possibly lead to a rapid takeover by the outstanding ones and a premature convergence to a weak generation. For the second situation, difficulty in converging, consider the scenario when the population converges to a set of chromosomes with similar fitness values, in this case, it will be hard to discriminate between chromosomes and converge.

To overcome these problems, one may use (1) a scaling procedure that uses a linear transformation of the objective function value to limit the competition early on, but to stimulate it later; or (2) a ranking procedure that ignores the objective function values.

Furthermore, a variety of functions could be used to discriminate between solutions, such as the original objective function of the optimisation problem under consideration, an auxiliary function or penalised objective function (similar to the choice of the form of the objective functions for the sequential metaheuristics).

II. Generic decisions for GA

As mentioned earlier, generic decisions are concerned with the parameters of the algorithm itself. Generic decisions for GA are (1) population size and selection of the initial population, (2) selection mechanism, (3) reproduction mechanism, (4) genetic operators' rates, (5) replacement mechanism and (6) stopping criteria. These decisions are presented hereafter.

(1) Population size and selection of the initial population

As it is shown in Table 4, the first problem-specific decision is related to the population that has a great influence on the GA's performance and speed. GA starts with an initial population of size M. There have been many studies on the optimal population size such as Goldberg (1989), Alander (1992) and Roeva et al. (2013). A small population might not provide enough room for different parent combinations to take place effectively and might result in the generation of solutions that bear close structural resemblance. This loss of diversity in the population affects the breadth of the search; that could increase the risk of seriously under-covering the solution space. On the other hand, a large population size could result in a disproportionate increase in the execution time of the algorithm without a substantial improvement in the quality of the solutions generated and the diminishing efficiency of the GA. Thus, when choosing the population size, one should make a trade-off between efficiency and effectiveness.

Another decision related to the population is the criterion to initialise the population. This criterion greatly affects the performance and speed of the GA algorithm. A common population initialisation is a random generation (Katayamaet al., 2000; Qu and Sun 1999). Additionally, the initial population can be supplied by a heuristic (Liao, 2009; Ray et al.; 2007; Kaur and Murugappan, 2008), using a gene bank (Wei et al. 2007), and a sorted population (Yugay et al., 2008), etc. An analysis of their performance is done by Paul et al. (2015) and Shanmugam et al. (2013). They found that "seeding" the population with a high-quality solution can help the GA find better

solutions rather more quickly than it can from a random start. However, there is a possible disadvantage in that the chance of premature convergence may be increased.

(2) Selection mechanism

Selection operators can be used throughout a chromosomes life cycle; namely, breeding, life and death. GA iteratively selects a group of chromosomes of the current generation for mating and another group for mutation. In addition, selection operators select a chromosome to be replaced by the new offspring, meaning that the chosen chromosome's life cycle has ended. One can use a single selection mechanism throughout the GA's process or a different selection mechanism for each stage of the chromosome's life cycle.

Most common selection mechanisms are related to a chromosome's fitness. A simple way of choosing parents to be paired in each generation is based on random or biased random sampling from the population; for example, parents may all be selected randomly, randomly on a fitness basis, or some parents may be selected randomly while others are selected on a fitness basis. A typical selection criterion gives a higher priority to fitter individuals since this leads to a faster convergence of the GA. However, if parents are selected randomly, this will give an equal probability of selection to each individual in the population. The most commonly used selection mechanisms are ordinal selection, proportional selection, ranking selection and steady-state selection (Baker, 1987; Goldberg, 1989; Mühlenbein and Schlierkamp-Voosen, 1993). For more detail, see Appendix E.

(3) Reproduction mechanism

A pair of chromosomes from the selected group (parents) reproduce new offsprings. The *crossover operator* is designed to exchange some genes from the parents' chromosomes in a structured yet randomised manner, hoping that offsprings inherit the good genes and perform better than their parents had. This operator exchanges information between individuals of the population and passes on the collected information to the next generation.

The issue with crossover operator is that in some cases some or all chromosomes become similar. In other words, all genes in chromosomes will be the same. To

overcome this issue, *mutation operator* randomly changes or modifies the genes in chromosomes to diversify the population. These operators are explained hereafter.

Crossover operator

This operator recombines a pair of chromosomes to create new offspring, preferably different from the parents. Many of the proposed crossover operators are problem-specific and mostly depend on the problem representation. The problem independent crossover operators are simple or one-point crossovers, multi-point crossovers, uniform crossovers and three-parent crossovers (Syswerda, 1989; Spears and De Jong, 1995; Sivanandam and Deepa, 2007). For more details, see Appendix F.

Mutation operator

Mutation operator is designed to diversify the search occasionally and lead the search out of local optima. This operator introduces new information to the population by randomly changing a gene or multiple genes of a chromosome.

(4) Genetic operators' rates

GA's crossover and mutation rates have a significant influence on GA's performance. Crossover rate (P_c) specifies the rate which the crossover operator is applied to create offspring. By controlling this rate, one can control the rate new individuals are created from one generation to the next. In other words, GA with a high crossover rate increases the diversity of the population by creating more offspring in each generation, conversely, GA with a lower rate has less diversity by creating fewer offspring but keeps the population information for the next generation.

On the other hand, mutation rate (P_m) specifies the probability as to which mutation operator is applied to modify a chromosome. As mentioned previously, a mutation operator is designed to diversify the population, whereas a mutation rate defines the rate of diversification. A high mutation rate increases the diversification level and helps guiding the search out the local optima. However, a high mutation rate could also cause the loss of information and lead to a random search.

Most GA implementations have assumed that the probability or rate of using a particular operator is fixed at the outset and remains the same throughout; e.g., a

mutation is applied with a low probability while crossover is applied with a high probability, the most common settings are shown in Table 5.

Author	P_c	P_m	M
De Jong (1975)	0.6	0.001	60
Grefenstette (1986)	0.95	0.001	30

Table 5 Static rates

However, to prevent the GA from premature convergence, one might make use of a procedure where the GA's parameters change during the process, some of these procedures are reviewed in the study by Daridi et al. (2004).

Author	Dependence
Schaffer and Morishima (1987)	Performance of the produced offspring
Fogarty (1989)	Time
Hesser and Männer (1991)	Population size and chromosome's length
Srinivas and Patnaik (1994)	Fitness value
Bäck and Schütz (1996)	Time, chromosome's length and maximum number of generations
Daridi et al. (2004)	chromosome's length and life time

Table 6 Adaptive GA

Adaptive GAs (AGA) do not require prespecified parameter value since these parameters are determined by the GA, see Table 6. Moreover, self-adaptation could improve the GAs performance significantly (Bäck, 1996; Daridi et al., 2004).

(5) Replacement mechanism

New offsprings should be introduced into the population by replacing an existing chromosome. In other words, the new generation is created by replacing a chromosome from the previous generation with a new offspring hoping that the new generation's quality, on average, is better than the previous one. In the replacement mechanism, a selection criterion is used to delete a chromosome from a previous generation and end its life cycle. A default criterion could be random selection of chromosomes to be deleted. The most common selection criterion used in replacement (Smith and Vavak 1999, Mumford 2004) are outlined as follows:

Delete oldest or worst: based on this criterion the oldest or worst chromosome is selected to be deleted from the generation.

Delete parents: since the parents' genes have been passed on to their offsprings, one might choose to delete the parents. However, in some cases this replacement mechanism could lead to loss of information about the order of genes.

Delete-all: this selection criterion selects all chromosomes from the previous generation to end their life cycle and replaces them with new offspring. Since the parents' genes have been passed on to their children, the information they had will not be lost, however, if only a subset of the previous generation had been selected for mating, the genes of chromosomes that were not in the mating pool will be lost. On the other hand, this selection criterion is parameter-free and easy to implement.

Steady-state: as it was mentioned before, this selection mechanism selects a subset of chromosomes to reproduce and a subset to be replaced with new offspring, where both subsets' size is equal and a parameter that should be specified. In addition, one should specify the two selection criteria, one for mating selection criterion and the other is for the replacement criterion.

Replacement-with-no-duplicates: based on this criterion, whichever selection mechanism is used, the criterion should also check the new offspring is not a repetition of an existing chromosome.

Moreover, there is no guarantee that the best member of a population will survive from one generation to the next, except for deleting the worst chromosome. To overcome this issue, one may use the elitism strategy. In the elitism strategy, GA is not permitted to delete the best member of the current population.

(6) Stopping criteria

Several stopping criteria can be used in genetic algorithms, such as a predetermined number of iterations or generations is reached; the computational time exceeds a predetermined time limit; the best objective function value found so far is not updated for a predetermined number of generations; a measure of the population diversity falls below a pre-specified threshold, etc. (Safe et al., 2004; Aytug and Koehler, 1996).

2.6. Hyperheuristic

Although tailor-made methods can produce good solutions within a reasonable time, they are usually limited to a particular type of problem. Since real world problems are likely to change over time, heuristics or metaheuristics might produce poor solutions or none at all. Moreover, some heuristics can deliver good solutions at certain but not all points. In other words, they might produce good quality solutions for some instances, but not all, of a specific problem. Thus, one can use a higher-level methodology to select or generate heuristics to solve COPs (Chakhlevitch and Cowling, 2008; Burke et al., 2010). These methodologies are call Hyperheuristics.

Fisher and Thompson (1963) birthed the idea behind hyperheuristics in the 1960s. In 1997, Denzinger et al. used the term 'hyper' in their technical report. They designed a method that combines several artificial intelligence algorithms resulting in an automated theorem prover. In 2000, Cowling et al. used the term 'hyperheuristic', and later Cowling et al. (2000, 2002a, b, c) developed the ideas behind hyperheuristic and applied it to scheduling problems.

At first, the term hyperheuristic (HH) was used to describe "heuristics to choose heuristics" (Cowling et al., 2000). Chakhlevitch and Cowling (2008) defined hyperheuristics as high-level heuristics that manage a set of low-level heuristics to find or design a good solution method for a COP by only making use of limited problem-specific information. On the other hand, Burke et al. (2010) defined hyperheuristics as automated methodologies for selecting or generating heuristics to solve hard computational search problems. Later, Burke et al. (2013) defined hyperheuristics as search methods or learning mechanisms for selecting or generating heuristics for COPs.

2.6.1. Hyperheuristic Classification

One of the hyperheuristic classifications by Chakhlevitch and Cowling (2008) classified hyperheuristics into four categories, namely hyperheuristics based on the random selection, greedy and peckish hyperheuristics, metaheuristic-based

hyperheuristics and hyperheuristics employing learning mechanisms to manage low-level heuristics, see Table 7. A summary of each category is as follows:

Hyperheuristics	Random Selection	Pure Random
		Random Descent
		Unbiased Random Process
		Monte Carlo
		Random With Deterministic Acceptance
	Greedy And Peckish	Accept Only Improving LLH
		Allow Non-improving LLH
	Metaheuristic-based	Genetic Algorithm
		Simulated Annealing
		Tabu Search
		Variable Neighborhood Search
	With Learning Mechanisms	Reinforcement Learning
		Learning Classifier System
		Case Based Reasoning
		Choice Function

Table 7 Chakhlevitch and Cowling (2008) hyperheuristics classification

I. Random Selection

This type of hyperheuristic is based on the random choice of low-level heuristic. Given a set of low-level heuristics, a random LLH is applied to the problem, although it might not produce a better solution. This search strategy is fast and straightforward, but it does not guarantee a better solution (Chen et al., 2016). However, modifications of random search and hybridizing it with other techniques, such as more advanced move acceptance techniques, could lead to better performance.

II. Greedy and Peckish

Greedy based hyperheuristics, in each stage of the search, select locally optimum LLH with the hope of finding the global optima. Peckish search strategies are a modification of greedy search strategy, which in each stage chooses LLH from a candidate list of the best neighbours of current LLH. These search strategies are time consuming and do not guarantee to find the best solution.

III. Metaheuristic-based hyperheuristic

In metaheuristic-based hyperheuristic, a metaheuristic is used as a high-level search strategy that guides the search away from local optima and efficiently selects the best or close to the best LLH. Variants of metaheuristic-based hyperheuristics has been proposed; such as hyperheuristics based on genetic algorithms (Fang et al.,1994; Hart et al., 1998, 1999; Cowling et al., 2002; Han et al., 2002; etc.), simulated algorithm (Bai and Kendall, 2003; Storer et al., 1995; Soubeiga, 2003) , tabu search (Storer et al., 1995; Burke et al., 2004, 2005; Burke and Soubeiga, 2003; etc.) and variable neighbourhood search (Qu and Burke, 2005; Chen et al., 2016).

GA-based hyperheuristic operate similar to traditional GA, although they have some differences. Traditional GA's search space is the problem space; however, GA-based hyperheuristic's search space is a set of LLH. In traditional GA, a chromosome represents a solution to the problem. On the contrary, GA-based hyperheuristic makes use of indirect presentation of chromosomes. A GA with the indirect encoding of the chromosome, instead of representing the solution itself, represents how the solution is solved; for example, a chromosome could represent a sequence of LLHs or parameters of a single LLH. GA-based hyperheuristic evolves these chromosomes to find better chromosomes.

Hyperheuristics based on SA, TS and VNS have similar search strategy as the traditional SA and TS. SA and TS based hyperheuristic s search the neighbourhood of current LLH and decide whether to accept or reject the new neighbour. On the other hand, *VNS-based hyperheuristics* use a set of neighbourhood structures to search the LLH search space.

IV. Hyperheuristics with Learning Mechanisms

As it is mentioned before**,** a learning mechanism is used to gather historical data, about the search space and hyperheuristic's performance, to select a promising LLH. Examples of learning mechanisms are choice function, reinforcement learning, learning classifier system and case-based reasoning.

Several other classification and categories are proposed such as Soubeiga (2003), Bai (2005) and Ross (2005), Bader-El-Den and Poli (2007), Burke et al. (2010, and 2013). For more detail on these classifications and categories see Appendix G.

2.6.2. Hyperheuristic specifications

The goal of designing a hyperheuristic method is finding an optimal or near optimal (sequence of) low-level heuristic(s) or component(s), depending on the nature of the hyperheuristic and its search space. As it is shown in the general hyperheuristics framework, hyperheuristics makes use of a search strategy to search the neighbourhood of the current LLH for a better neighbour, which could be using a learning mechanism or not, and decides whether to accept or reject the neighbour based on acceptance criterion. Thus, we can categorise the hyperheuristic specifications into HH nature, search space nature, HH search strategy, acceptance criteria and learning mechanisms, see Figure 3.

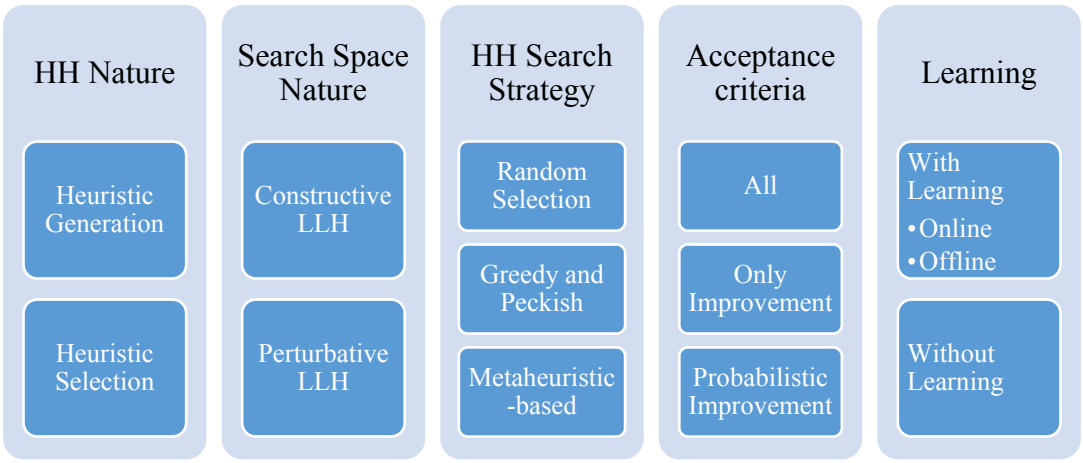


Figure 3 Hyperheuristic specifications

Since any combination of the HH nature, search space nature, HH search strategy, acceptance criteria and learning would lead to an HH category where each can have different performance, upsides and downsides. Thus, one can make use of several components to implement a better HH to overcome a single category's limitation and design a general framework for hyperheuristics, see Figure 4. Note that, the implementation decisions of hyperheuristic are the choice of the hyperheuristic specifications.

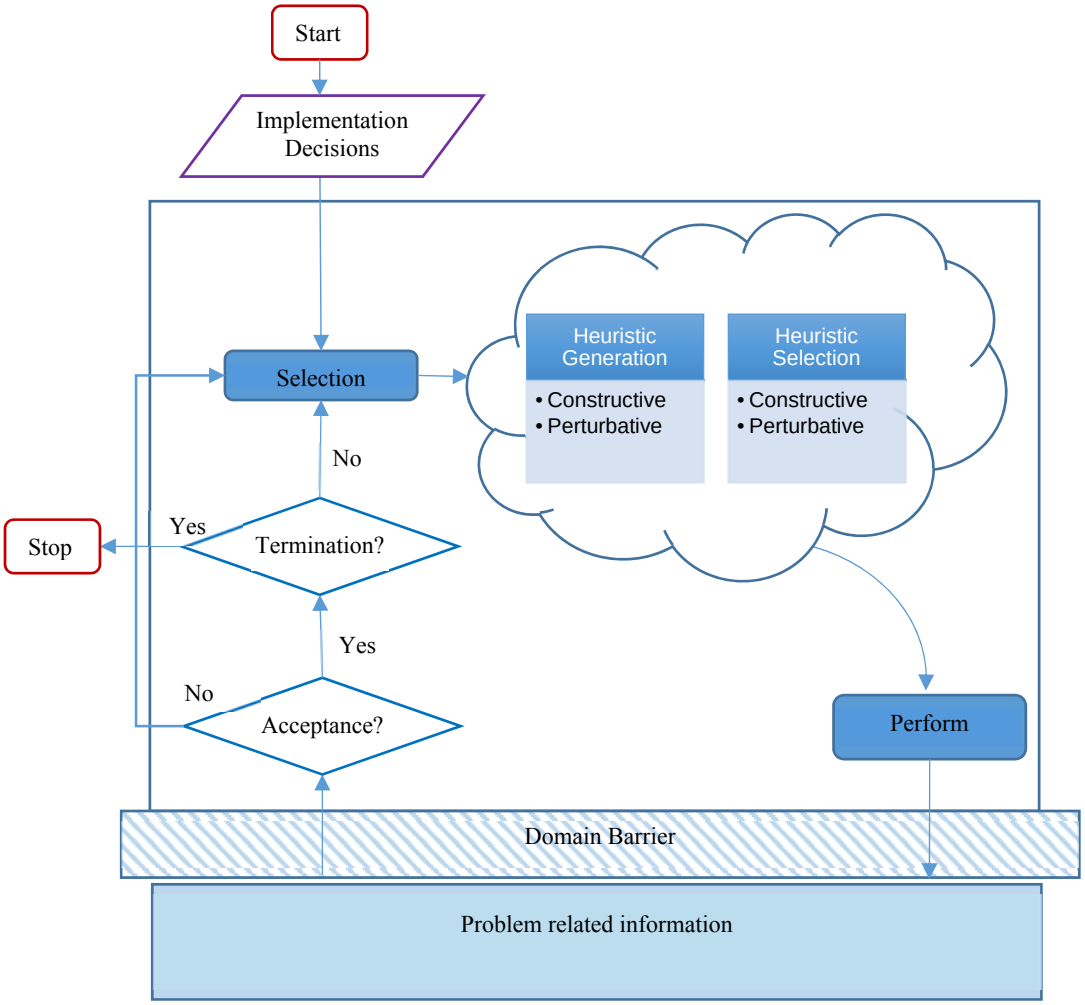


Figure 4 Hyperheuristic framework

2.7. Local search in the space of infeasible solutions

Up to this point in time, most of the published literature is on local search methods in the feasible space, whether classical local search, metaheuristics or hyperheuristics, with few exceptions searching in the feasible-infeasible solution space. The first search methods start and explore the search within the feasible space, without allowing the search to leave the feasible space. Similarly, the second search methods start the search from within the feasible space, however, they allow the search to temporarily leave the feasible space. On the other hand, one can search start from and explore the infeasible space, each time reducing infeasibility, and progress towards the feasible space. This method was first proposed, called DLS, by Ouenniche and his collaborators (Ouenniche et al., 2017) to solve the TSP by exploring its infeasible space.

As compared to primal local search-based heuristics' designs, whether classical local search or metaheuristics, Ouenniche et al. (2017) integrated design features of optimal algorithms. DLS starts the search with an infeasible solution (e.g., the solution of a relaxation of the problem under consideration) and progresses towards a feasible solution by using new neighbourhood structures to repair the intermediate infeasible solutions. Moreover, to prevent exploring search areas with no potential of good solutions they used a bounding scheme. Ouenniche et al. (2017) also made a conceptual comparison between DLS and B&B, see Table 8. For more details, refer to Ouenniche et al. (2017).

B&B	**DLS**
Break one sub-tour at a time	Break two or more sub-tours at a time
Examine all possible ways of excluding or including one edge at a time of one sub-tour	Examine all possible ways of breaking two or more sub-tours (e.g., breaking one or more edges in each sub-tour) and connecting them
At each level of the B&B tree – except level 0, exclude (resp., include) one edge of one of the sub-tours and keep all the remaining edges free except those fixed at higher levels of the tree, if any	Exclude two (or more edges), one (or more edges) from each sub-tour, and include in the next solution all the remaining edges in the sub-tours
At each node of the tree, a re-optimization process is invoked, which could lead to a new infeasible or feasible solution	At each node of the tree, a ''restricted'' optimization process is invoked, which could lead to a new infeasible or feasible solution, but with potentially more similarity to the solution of the parent node as compared to B&B. To diversity in terms of structure of partial solutions and explore more nodes as done in B&B, we use a second type of moves similar in spirit to branch exchange improvement
A new branch would not necessarily lead to a reduction in the number of sub-tours	Each infeasible neighbourhood move systematically reduces the number of sub-tours by one or more
Break one sub-tour at a time	Break two or more sub-tours at a time

Table 8 Comparative analysis between B&B and DLS

This thesis refines and extends Ouenniche et al (2017) proposed infeasible search framework. We propose a generic and parameterized local search in the space of feasible space (GPILS) as a refinement of the DLS framework proposed by Ouenniche et al (2017), where we customise GPILS to solve the TSP.

2.8. Conclusion

In this chapter, the most common and relevant mathematical formulations, properties and relaxations of the TSP, descriptions of the solution methodologies and state-of-the-art techniques are presented. We classified the heuristic solution approaches into three categories; namely, feasible (primal), infeasible-infeasible and infeasible methodologies. Most research have been done in the first two categories, however, to the best of our knowledge there is still not much work done in the third category, i.e. infeasible methodologies.

In the rest of this thesis we shall contribute to the methodologies in the infeasible search space by refining and enhancing the work done by Ouenniche et al (2017). We shall propose an enhanced framework for local search in the infeasible space and automation of the choice of its parameters using a hyperheuristic framework. We also investigated the reusability of the proposed methodology for unseen (new) problem instances.

3. A Generic Parameterised Infeasible Local Search Framework

So far, most of the published literature on local search methods, whether classical local search, metaheuristics or hyperheuristics, starts with a feasible solution and only searches in the feasible solution space to find a better solution. Note however that there are few exceptions where the search can move into the infeasible space, but it is forced to move back to the feasible space. On the other hand, one can start from and explore the infeasible space, each time reducing infeasibility, and progress towards the feasible space. Ouenniche and his collaborators (Ouenniche et al., 2017) were the firsts to propose such methods, called dual local search (DLS), to solve the TSP by exploring its infeasible space. DLS is a local search framework designed to solve COPs so that the search starts within the space of infeasible solutions and progresses towards the space of feasible solutions. Once the infeasible search lands in the feasible space, one could choose to either end the search or continue exploring the feasible space. When the option of choosing to explore the primal space is chosen, the design becomes an infeasible-feasible local search. Note, however, that one could explore the primal space using either a primal methodology or a feasible-infeasible methodology.

The contribution of this chapter is to investigate the possibility of starting from an initial infeasible solution and progress toward the feasible space by searching the infeasible solution space. Therefore, in this chapter, we shall extend and refine DLS proposed by Ouenniche et al. (2007) and propose a generic parameterised infeasible local search algorithm referred to as GPILS and discuss the rationale behind it. The proposed framework is stated to accommodate any COP. For illustration purposes, we shall provide a customised version for the TSP. Then, we shall discuss implementation

decisions and the empirical analysis. Later, we shall present the conclusion and final remarks of this chapter.

The rationale behind the design of a generic parameterised infeasible local search (GPILS) is to integrate the design features of optimal algorithms (e.g., branch-and-bound) into local search. To be more specific, the features of GPILS borrowed from optimal methodologies include starting with an infeasible solution, making use of a bounding scheme to prevent exploring search areas with no potential for good solutions and using infeasible neighbourhood search structures that exploit exact methods features such as branching rules.

In this chapter, we divide the moves of such infeasible neighbourhood search structures into two categories; namely, Type I moves and Type II moves, where Type I moves define a partial "repair" mechanism for infeasible solutions, and Type II moves define a local improvement mechanism of components of infeasible solutions. In analogy with optimal solution methodologies; e.g., branch-and-bound, we make use of Type II moves to avoid under-exploring the search tree, or equivalently to allow exploring search tree branches that would be unexplored otherwise. The flowchart of the proposed GPILS framework is provided in Figure 5, and a detailed generic pseudo-code suitable for any given COP is provided afterwards in Table 9.

The proposed infeasible search framework could be adapted to solve any combinatorial optimisation problem. In the following subsections, a customised version is provided for solving the TSP; see Table 10 for a detailed generic pseudo-code customised for solving the TSP. Notice that unlike DLS, within our GPILS, we make use of a recursive function to explore the infeasible neighbourhood. The rationale behind this choice is discussed in the next section. Hereafter, we shall discuss the implementation decisions of the proposed infeasible search framework.

3.1. Initialisation of the bounding scheme and the seed

As it was mentioned previously, the infeasible local search starts the search with an infeasible solution and progress towards the primal solution using a new

neighbourhood structure, while pruning the nonpromising search areas considering abounding scheme. The bounding scheme consists of a primal bound and a dual bound.

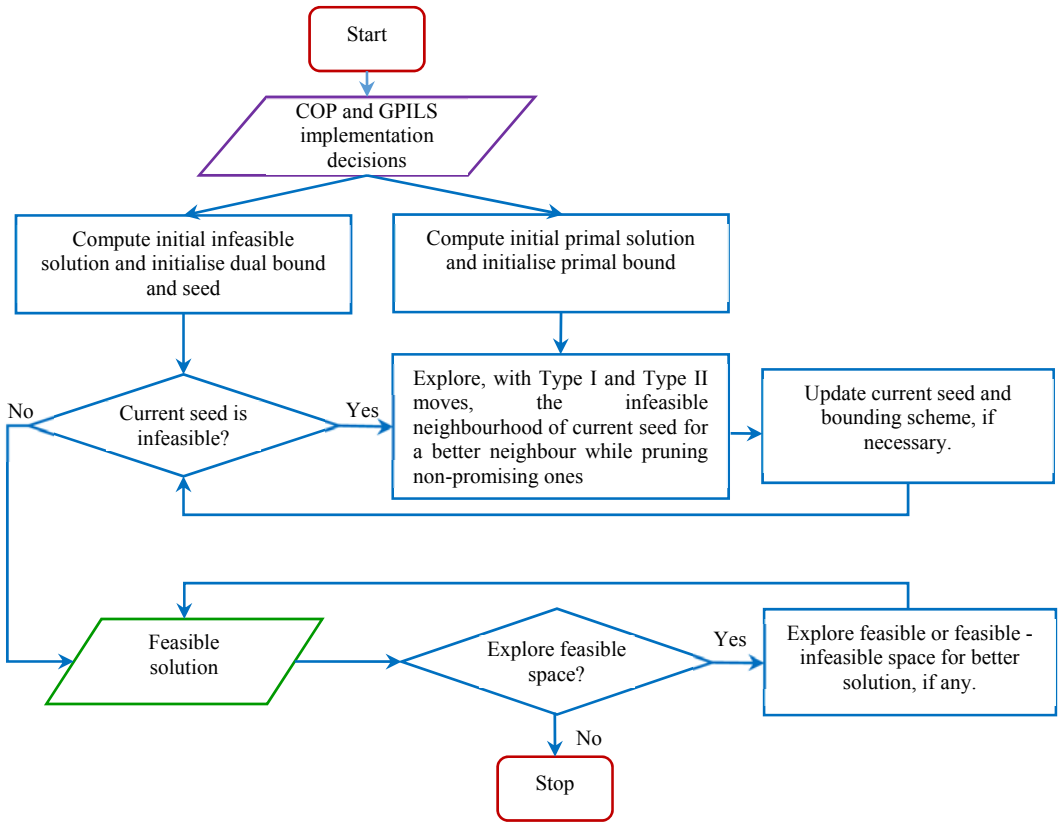


Figure 5 Flowchart of the GPILS framework

In order to initialise the primal (upper) bound, one has to choose a primal method, say PM, to use for obtaining a primal solution or a feasible tour, say x, to the TSP and use it to initialise the primal bound (PB); i.e., set PB to the objective function value or total distance of x. In this thesis, the choice of the method to devise a primal solution for initialising the primal bound is represented by the categorical variable PM, where a default category could involve using a randomised procedure, some categories could correspond each to a different construction heuristic for the TSP (e.g., nearest neighbour procedure, Clarke and Wright savings procedures, insertion procedures, nearest merger procedure), and other categories could correspond to any combination of a construction heuristic and a primal local search method for its improvement (e.g., construct a TSP tour using the nearest neighbour procedure and improve it using simulated annealing).

Input
An instance of the COP under consideration.

Output
Feasible or primal solution to the COP under consideration.

Implementation Decisions
// Choose how to Initialise the bounding scheme and the seed
Choose a primal method, say PM, to use for obtaining a primal solution, say x, to the COP under consideration;
Choose a infeasible method, say IM, to use for obtaining a infeasible solution, say y, to the COP under consideration;
// Choose how to explore the infeasible space
Choose the repair mechanism, or equivalently the Type I moves ($T1M$), to use in exploring the infeasible space of COP as well as the performance metric to be used for assessing infeasible neighbours ($IMetric$);
Choose the local improvement mechanism, or equivalently the Type II moves ($T2M$), to use in improving components of the infeasible neighbours generated by Type I moves;
Choose the design of the infeasible neighbourhood search structure to use in exploring the infeasible space of COP (*infeasible_neighbourhood_structure*). One of two alternatives could be chosen. The first option is to use Type I moves to explore the infeasible space and select the best neighbour, then Type II moves are used to improve the best neighbour locally – we refer to this design as improve the best neighbour (IBN) design. The second option is to immediately use Type II moves to improve every neighbour obtained with Type I moves – we refer to this design as improve all neighbours (IAN) design.
// Choose whether to explore the primal space and how
Choose whether to explore the primal space or not (*explore_primal_space*). When the option of exploring the primal space is chosen; i.e., *explore_primal_space* = 1, one has to choose the primal neighbourhood structure to use (*primal_neighbourhood_structure*), the improvement mechanism (*primal_improvement_mechanism*) as well as the performance metric to be used for assessing primal neighbours ($PMetric$).

Initialisation Step
// Initialise the bounding scheme and the seed
Use PM to initialise the primal bound (PB); i.e., set PB to the value of the objective function of the COP under consideration evaluated at x, say $z(x)$;
use IM to initialise the dual bound (DB) and the seed, say y_{seed}; i.e., set DB to the objective function value of y, say $z(y)$, and set the seed y_{seed} to y;

Iterative Step
WHILE current seed is an infeasible solution to COP **DO** {
    Explore the infeasible neighbourhood – as specified by Type I moves, Type II moves and infeasible neighbourhood search structure – of the current seed y_{seed} for a better neighbour, while pruning non-promising infeasible neighbours, and update the current seed;
    Update the bounding scheme (PB, DB), if necessary;
}
IF explore_primal_space = 1 **THEN**
    Use a primal or a feasible-infeasible local search framework to explore the primal space for a better solution, if any, according to the choices made through *primal_neighbourhood_structure*, *infeasible_neighbourhood_structure*, *primal_improvement_mechanism*, and *PMetric*;

Table 9 Pseudo-code of the proposed GPILS framework

Input

Instance of the TSP; i.e., the distance or cost matrix, say C.

Output

Feasible or primal solution to the TSP.

Initialisation Step

// Initialise the bounding scheme and the seed – section 2.1 for more details

Choose a primal method, say PM, to use for obtaining a primal solution, say x, to the TSP and use it to initialise the primal bound (PB); i.e., set PB to the total cost of the TSP tour x, say $z(x)$;

Choose a infeasible method, say IM, to use for obtaining a infeasible solution, say y, to the TSP and use it to initialise the dual bound (DB) and the seed, say y_{seed}; i.e., set the seed y_{seed} to y, and set DB to the total cost of y, say $z(y)$;

// Choose how to explore the infeasible space – section 2.2 for more details and implementation decisions

Choose whether or not to exploit domain knowledge to enhance the efficiency and/or the effectiveness of the search (*exploit_domain_knowledge*).

IF *exploit_domain_knowledge* is set to 1, **THEN** the following repair and local improvement mechanisms, or equivalently Type I and Type II moves, should be candidate set-based, where a candidate set, say CS, refers to a subset of the set of possibilities to perform Type I and Type II moves defined so as to narrow down the search space to "promising" regions as specified by a set of criteria that exploit the domain knowledge of the TSP instance under consideration;

Choose the repair mechanism, or equivalently the Type I moves ($T1M(.)$), to use in exploring the infeasible space of the TSP as well as the performance metric to be used for assessing infeasible neighbours ($IMetric$):

$T1M$(*breaking_method(.), patching_method(.),* $IMetric$),

where Type I moves consist of two main operations; namely, a breaking operation:

breaking_method (s, *subtours_selection_criterion*, r, *arcs_to_break_selection_criterion*),

moreover, a patching operation:

$patching\_method(C, start\_with\_subtour,\ paths\_to\_merge\_selection\_criterion,$
$paths\_to\_patch\_selection\_criterion, merging\_criterion, patching\_Criterion$, n_{paths}^{patch}, n_{paths}^{insert},
$type\_of\_patching\_operation, patching\_operation\_performance\_criterion,$
$type\_of\_implementation)$

in sum, one has to choose the initial parameters of these functions – see section 2.2 for more details;

Choose the local improvement mechanism, or equivalently the Type II moves ($T2M$), to use in improving components of the infeasible neighbours generated by Type I moves;

Choose the design of the infeasible neighbourhood structure to use in exploring the infeasible space of the TSP (*INS*). One of two alternatives could be chosen. The first option is to use Type I moves to explore the infeasible space and select the best neighbour, then Type II moves are used to improve the best neighbour locally – we refer to this design as improve best neighbour (IBN) design. The

second option is to immediately use Type II moves to improve every neighbour obtained with Type I moves, we refer to this design as improve all neighbours (IAN) design.

// Choose whether to explore the primal space and how

Choose whether to explore the primal space or not (*explore_primal_space*). When the option of exploring the primal space is chosen; i.e., *explore_primal_space* = 1, one has to choose the primal neighbourhood structure to use (*PNS*), the improvement mechanism (*primal_improvement_mechanism*) as well as the performance metric to be used for assessing primal neighbours ($PMetric$).

Initialise iteration counter, say k, to 1;

Initialise the number of subtours to break and merge at iteration k, say s^k, to s;

Initialise the number of edges to break in each subtour S_i^k at iteration k, say r_i^k, to r;

Iterative Step

WHILE current seed is an infeasible solution to the TSP **DO {**

// Explore the infeasible neighbourhood – as specified by Type I moves, Type II moves and

// infeasible neighbourhood structure – of the current seed y_{seed} for a better neighbour, while

// pruning non-promising infeasible neighbours

Choose the set, say $\{S_i^k\}$, of subtours of the current seed, y_{seed}, to break and patch at a time, where the specific set of subtours $\{S_i^k\}$ is chosen based on *subtours_selection_criterion*;

Choose the set of candidate edges to break, say CS_i^k, for each subtour i to be used at iteration k, $i = 1, \dots, s^k$, based on *arcs_to_break_selection_criterion* and initialise the array of indexes of edges to break in each subtour to the empty set; i.e., set $e_{S_i^k} = \emptyset$ for $i = 1, \dots, s^k$;

Initialise subtour index counter m to 1, edge index counter ℓ to 1, and loop ℓ initialiser j_m^ℓ to 1;

// Call the recursive infeasible neighbourhood search function to search for the best neighbour of

// y_{seed}, say y_k^* – see Table 24 for a detailed pseudo-code

y_k^* = RINS $(s^k, \{S_i^k\}, \{r_i^k\}, \{CS_i^k\}, \{e_{S_i^k}\}, m, \ell, j_m^\ell, C, INS, T1M(.), T2M, IMetric,$ $IMetric(y_k^*), patching\_method)$;

// Improve the current infeasible solution y_k^* locally using Type II moves, if required

IF *INS* = *IBN* **THEN** y_k^* = *PerformTypeIIMove*$(T2M, y_k^*)$;

Increment iteration counter k by 1; that is, set $k = k + 1$;

Update the current seed, y_{seed}; i.e., set $y_{seed} = y_k^*$;

Update the bounding scheme (PB, DB), if necessary;

}

IF *explore_primal_space* = 1 **THEN**

Use a primal or a feasible-infeasible local search framework to explore the primal space for a better solution, if any, according to the choices made through *PNS*, *INS*, *primal_improvement_mechanism*, and *PMetric*;

Table 10 Pseudo-code of the proposed GPILS framework for TSP

As to initialising the dual bound and the seed, one has to choose a infeasible space-based method, say IM, to use for obtaining a infeasible solution, say y, to the TSP and use it to initialise the dual bound (DB) and the seed, say y_{seed}; i.e., set DB to the objective function value or total distance of y, say $z(y)$, and set the seed y_{seed} to y. An infeasible solution y to the TSP instance under consideration is a set of subtours $\{S_i^k, i = 1, \dots, N_{sbt}\}$, where S_i^k denote the i^{th} subtour at iteration k and N_{sbt} denote the number of subtours in the infeasible solution. Note that $S_1^k, \dots, S_{N_{sbt}}^k$ cover all TSP nodes.

The initial infeasible solution y could be obtained in different ways. In this thesis, the choice of the method to generate the initial infeasible solution is represented by a categorical variable IM, where a default category could involve using a randomised procedure, some categories could correspond each to a different relaxation of a TSP formulation (e.g., AP-based relaxation of the TSP), and other categories could correspond to some rule-based heuristics or clustering methods, see Appendix H, to allow for exploiting the structure of a TSP instance (e.g., $\mathcal{K}$-means).

With respect to rule-based heuristics, we propose a *parameterised infeasible heuristic* which we call PIH where the parameters are the number of subtours N_{sbt}, the sizes of subtours $|S_k|, k = 1, \dots, N_{sbt}$, a decision rule DR for assigning nodes to subtours or clusters, unless the nodes are chosen randomly, a heuristic for constructing a tour S_k that visits each node in cluster k once and only once, and an option for locally improving the subtours, see Table 11for the pseudo-code of $PIH(CO;DRA;DRA_k, k = 1, \dots, N_{sbt};N_{sbt};|S_k|, k = 1, \dots, N_{sbt};DRC;Imp;\ IM;NS)$.

To conclude this section, we would like to point out that PIH only requires slight modifications to generate a good quality infeasible solution for other routing problems. For example, through the decision rule for assigning nodes to subtour/cluster or a decision rule for constructing a tour one could address additional constraint such as capacity or time windows constraints. In future research, we intend to apply this method to other routing problems.

Choose whether to determine the clusters one by one or all at once. This decision is represented by a binary variable CO, where $CO = 0$ refers to the first option and, $CO = 1$ refers to the second option;

IF $CO = 0$ **THEN** {

Choose the number of subtours N_{sbt} in the infeasible solution;

Choose the sizes $|S_k|, k = 1, \dots, N_{sbt}$ of subtours in the infeasible solution;

Choose a decision rule DRA_k for assigning nodes to subtour or cluster $k,\ k = 1, \dots, N_{sbt}$. This decision is represented by a categorical variable DRA_k, where a default category could involve using a randomised procedure (i.e., nodes are arbitrarily assigned to clusters) and other categories could correspond each to a different criterion (e.g., nearest neighbour, farthest neighbour);}

ELSE {

Choose a decision rule for assigning nodes all at once to subtours or clusters. This decision is represented by a categorical variable DRA, where each category corresponds to a different clustering method). Depending on the choice of the clustering method, one might have to specify the number of subtours N_{sbt} in the infeasible solution or leave it to the clustering method; }

Choose a decision rule DRC for constructing a tour S_k that visits each node in cluster k once and only once. This decision is represented by a categorical variable DRC, where a default category could involve using a randomised procedure (i.e., random tour) and the remaining categories could correspond each to a construction heuristic;

Choose whether to improve the subtours of the infeasible solution locally or not. This decision is represented by a binary variable Imp. If this option is on; that is, $Imp = 1$, then the improvement mechanism IMP and the underlying neighbourhood structure NS should be specified, where IMP is a categorical variable representing the different options for the improvement mechanism (e.g., classical local search algorithms, metaheuristics), and NS is a categorical variable representing the different options for the neighbourhood structure (e.g., 2-opt, 3-opt).

IF $CO = 0$ **THEN {** //Determine clusters

FOR $k = 1\ to\ N_{sbt}$ **DO {**

Amongst the nodes not yet assigned to any cluster, assign $|S_k|$ nodes to cluster k according to the decision rule specified by DRA_k;}**}**

ELSE Then Determine clusters and their number, if necessary, according to the decision rule specified by DRA;

FOR $k = 1\ to\ N_{sbt}$ **DO** // Construct and eventually improve subtours **{**

Construct a tour S_k that visits each node in cluster k once and only once according to the decision rule specified by DRC;

IF $Imp = 1$ **THEN** Improve subtour S_k according to the improvement mechanism and the neighbourhood structure specified by IMP and NS, respectively; }}

Table 11 Pseudo-code of the parameterised infeasible-based heuristic (PIH)

3.2. Exploration of the infeasible space

Exploration of the infeasible space requires several decisions to be made. First, one must choose the repair mechanism, or equivalently the Type I moves, to use in exploring the infeasible space of the TSP as well as the performance metric to be used for assessing infeasible neighbours. Second, one must choose the local improvement mechanism of components of the infeasible neighbours generated by Type I moves, or equivalently the Type II moves. Third, one must choose the design of the infeasible neighbourhood structure to use in exploring the infeasible space of the TSP. These decisions are discussed hereafter.

3.2.1. Repair mechanism

A variety of repair mechanisms could be designed for use at this stage. We propose a generic and parameterised repair mechanism that involves two basic operations: (a) breaking a number of subtours and (b) patching the broken subtours to form a single larger subtour. The implementation of this repair mechanism of an infeasible solution requires a number of decisions to be made; namely:

- Specification of the number of subtours to break and patch at a time, say s;
- Specification of a selection criterion according to which subtours to break and patch at a time is chosen;
- Specification of the number of arcs to break in each subtour, as part of the break and patch or repair mechanism, say $r_k,\ r = 1, \dots, s$;
- Specification of the selection criteria according to which the arcs to break and those to add, as part of the break and patch operations, are chosen;
- Specification of a metric to use for measuring the performance of the break and patch or repair operation.

The decisions above are discussed hereafter.

3.2.2. Number of subtours to break and patch

The specification of the number of subtours to break and patch at each iteration, say s, is represented by a numerical variable and takes on integer values ranging from 2 to

the current number of subtours in the current infeasible solution, N_{sbt}. In our empirical investigation, we considered several values for s, see section 3.7. Note however that, from a computational perspective, a trade-off should be made between choosing relatively high values or relatively low values for parameter s. Choosing high values for parameter s would require a relatively small number of iterations to converge but would require exploring a relatively large number of possibilities for breaking s subtours and patching them. However, choosing relatively low values for parameter s would require a relatively large number of iterations to converge but would require exploring a relatively small number of possibilities for breaking s subtours and merging them. In our preliminary empirical investigation, we observed that, for most TSP instances, the initial infeasible solution obtained by solving an AP-based relaxation consists of a relatively large number of subtours of small cardinality (e.g., 2 and 3). In this case, a strategy that seems to deliver very good solutions consists of choosing a relatively large number of these small cardinality subtours at the beginning of the infeasible search process and decrease such number as the search progresses. In sum, in this case, a dynamic scheme for the choice of s^k seems to be a compromise between quality of the solution and the computational requirement. The above-mentioned observations should be exploited when a single instance of GPILS is implemented; that is, the analyst specifies all parameters of GPILS. In our empirical investigation, the parameters of GPILS were optimised using a metaheuristic.

3.2.3. Subtours selection criteria

The specification of the selection criteria according to which subtours to break and patch at a time is chosen is represented by the categorical variable *subtours_selection_criterion*. In our empirical investigation, we considered a relatively large number of categories including several "pure" categories where a "pure" category refers to one that makes use of a single criterion, shown in Table 12. Along with these categories a variety of hybrids where more than one criterion defines a category and are used sequentially to select subsets of subtours (e.g., first select s_1 largest subtours, then the remaining $(s - s_1)$ subtours are selected according to the closest subtour length criterion) can be considered.

As is shown in Table 12, the subtour selection criterion could be categorised as cost-based (shortest or longest), cardinality-based (smallest or longest), distance-based (closest or farthest) and merging cost-based criteria (cheapest or most expensive). Cost-based criteria require calculating the subtour cost, which could include travel cost, fuel cost, loading and unloading cost, etc. In the case of STSP, the subtours cost is the sum of travelled arc's weight. The cardinality-based criteria are based on the number of nodes in the subtour. The distance-based criteria (closest and farthest subtours) require calculating the distance between the subtours. Subtours distance matrix SD is used for distance-based criteria. In order to calculate SD one has to compute the minimum distance between all pairs of subtours $(S_k,\ S_\ell)$, where a subtour $S \in \bar{S}$ is specified by its cardinality $|S|$ and its sequence of nodes $(S(1),\ \dots,\ S(|S|))$, see pseudocode in Table 13 and example in Figure 6.

Paths to merge Selection Criteria	**Description**
Arbitrary	Random choice of paths
Cardinality-Based	Based on number of nodes in the subtour
Cost-Based	Based on travel cost/length of the subtours
Distance-Based	Based on the distance between the subtours
Merging Cost-Based	Based on the cost of merging each pair of subtours

Table 12 Subtours selection criterion

Input

$c = (c_{ij})$; $\bar{S}$: Subset of components or subtours of the infeasible solution $\bar{S} = \{S_i; i = 1, \dots, N_{sbt}\}$:

Output

SD: Subtours distance matrix $SD = (S_{kl})$

Iterative Step

FOR all pairs of subtours $(S_k,\ S_\ell)$, compute the minimum distance between subtour S_k and subtour S_l as follows:

$$SD(S_k,\ S_\ell) = min\{c(S_k(1), S_\ell(1)), \dots, c(S_k(1), S_\ell(|S_l| - 1)), \dots, c(S_k(|S_k| - 1), S_\ell(1)), \dots, c(S_k(|S_k| - 1), S_l(|S_l| - 1))\};$$

Table 13 Subtours distance matrix

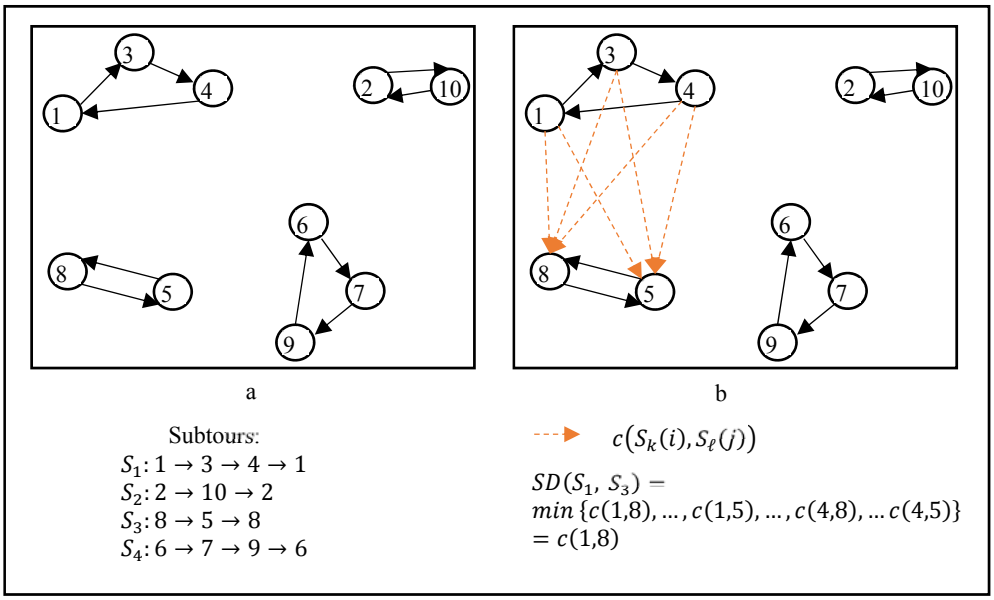


Figure 6 Subtours distance

Input

$c = (c_{ij})$; $\bar{S}$: Subset of components or subtours of the seed $\bar{S} = \{S_i; i = 1, \dots, N_{sbt}\}$:

Output

SMC: Subtours merging cost matrix, $SMC = (S_{kl})$

Initialisation Step

Initiate subtours merging cost matrix $SMC(S_k, S_l) = \infty$;

Iterative Step

FOR each pair of subtours ($\boldsymbol{S_k}$, $\boldsymbol{S_\ell}$), compute the minimum merging cost of subtour $\boldsymbol{S_k}$ and $\boldsymbol{S_l}$ as follows:

$Delete\_Edges\_Cost = c(S_k(i), S_k(i+1)) + c(S_l(j), S_\ell(j+1))$;

$Merging\_Cost\_Type1 = (S_k(i), S_l(j+1)) + c(S_k(j), S_\ell(i+1)) - Delete\_Edges\_Cost$;

$Merging\_Cost\_Type2 = (S_k(i), S_l(j)) + c(S_k(j+1), S_\ell(i+1)) - Delete\_Edges\_Cost$;

IF ($Merging\_Cost\_Type1 \leq Merging\_Cost\_Type2$ & $Merging\_Cost\_Type1 \leq SMC(S_k, S_l)$) **THEN**

$SMC(S_k, S_l) = Merging\_Cost\_Type1$;

Else IF ($Merging\_Cost\_Type1 > Merging\_Cost\_Type2$ & $Merging\_Cost\_Type2 \leq SMC(S_k, S_l)$) **THEN**

$SMC(S_k, S_l) = Merging\_Cost\_Type2$;

Table 14 Subtours merging cost matrix

Furthermore, merging cost-based criterion is based on the cost of merging a pair of subtours. For this criteria subtours merging cost matrix SMC is computed as shown in Table 14 and Figure 7.

3.2.4. Number of arcs involved in repair mechanism

The specification of the number of arcs to break in each subtour, as part of the break and patch or repair mechanism, say $r_k \leq |S_k|,\ k = 1, \dots, s$, is represented by a numerical variable and takes on integer values ranging from 1 to the cardinality of subtour S_k, $|S_k|$, when no constraints are imposed on the arcs to break; however, this upper bound could be lower / tighter when such constraints are imposed by previous decisions such as *arcs_to_break_selection_criterion* and *arcs_to_add_selection_criterion*. In our empirical investigation, we considered several values for r_k , see section on empirical results.

Note that a trade-off should be made between choosing relatively high values for parameters r_k $(k = 1, \dots, s)$ and low values. In fact, high r_k values would result in a relatively high-level of solution perturbation and would lead to a relatively large number of possibilities for connecting the broken subtours as well as a relatively large number of objective function evaluations, whereas low r_k values would result in relatively stable solution structures and keep the computational requirements relatively low. These observations should be exploited when GPILS is implemented and the analyst specifies its parameters. In our empirical investigation, the parameters of GPILS were optimised using metaheuristic. However, to keep the computational requirements reasonable, we set the upper bound on r_k.

3.2.5. Arcs selection criteria

The specification of selection criteria according to which the arcs to break and those to add, as part of the break and patch operations, are chosen is represented by two categorical variables; namely, *arcs_to_break_selection_criterion* and *arcs_to_add_selection_criterion*, respectively. A default selection criterion could be defined whereby no limitation is imposed on the choice of the arcs to break. In this case, the number of possible arcs to break and those to add is $\sum_{j=1}^{s^k} \binom{|S_j|}{r_j}$. Alternatively,

one can consider several categories for *arcs_to_break_selection_criterion* and *arcs_to_add_selection_criterion* using the concept of candidate sets, say CS, to limit the number of combinations with a prespecified cardinality, say K. To be more specific, the number of possible arcs to break and those to add will be reduced to $\sum_{j=1}^{s^k} \binom{K}{r_j}$.

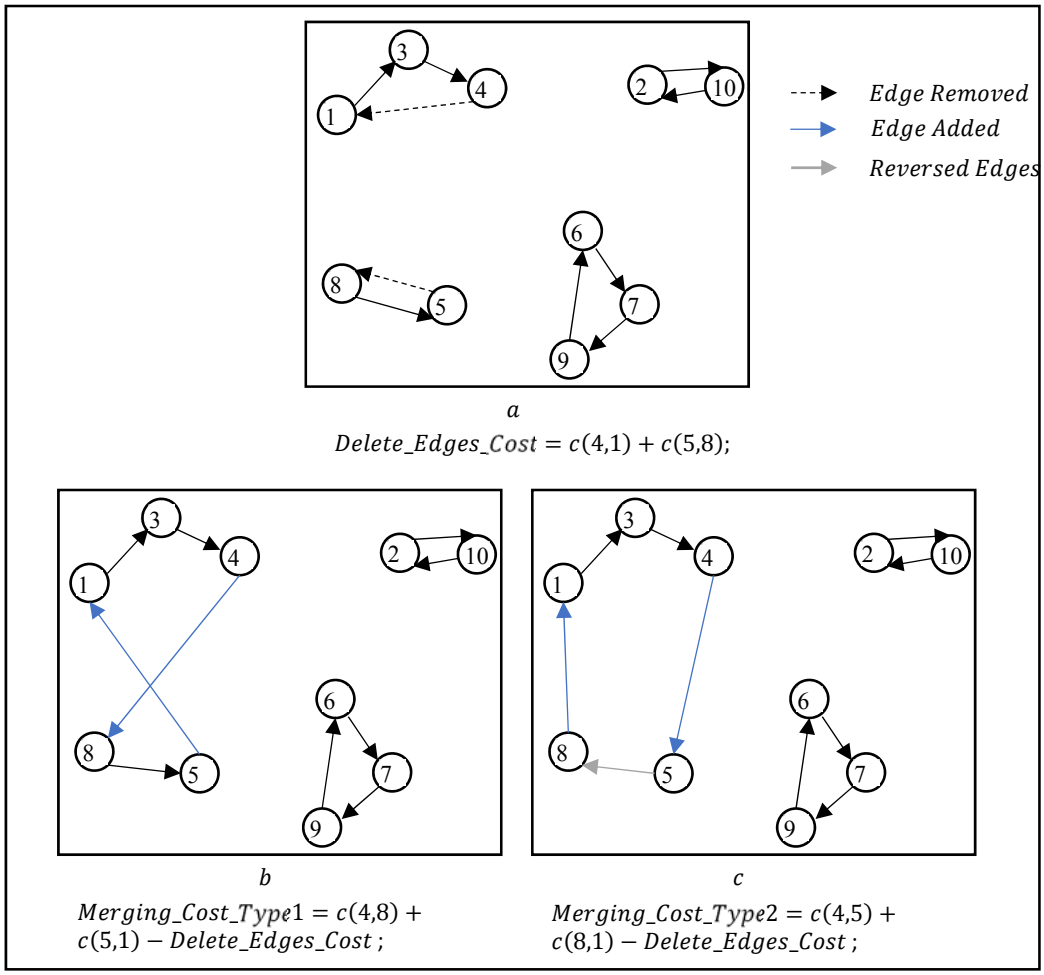


Figure 7 Subtours merging cost matrix

Note that more than one candidate sets could be used as categories, where these candidate sets or categories could be defined using a single criterion or multiple criteria. For example, one could define three candidate sets based on arc weight (e.g., travel cost, travel distance, travel time) as a single criterion along with two thresholds for arc weights chosen to reflect, for example, small, medium and large weights. A candidate set could be not to involve any previously added edges throughout the whole solution process or for a prespecified number of iterations. Alternatively, one could exploit the distribution of weights of arcs between pairs of nodes to setup the categories

or candidate sets. In our empirical investigation, we used K-NN and considered several values for K, see section 3.7.

As to the definition of the categories of *arcs_to_add_selection_criterion*, several categories could be defined; for example, a default category would not put any restriction on the possible set of arcs to use in connecting the broken subtours, a second third category would not involve any previously broken edges throughout the whole solution process; and a third category would not involve any broken edges throughout a prespecified number of iterations. Alternatively, one could use weight information to limit the number of possibilities of connecting the broken subtours. Once again hybrids, where more than one criterion defines a category, could be used. The implicit choice of the subset of arcs from which to select those to add for repairing a solution requires the selection of a patching method according to which arcs will be added to connect the broken subtours. Note that specific choices of *arcs_to_break_selection_criterion* and *arcs_to_add_selection_criterion* could influence one another and therefore would need to be consistent. For example, if the criterion or criteria chosen for the selection of arcs to break are less (respectively, more) restrictive than those for the selection of arcs to add, one might face a situation where the repair operations cannot be performed. In sum, the moves in the parameter space of GPILS should be chosen to avoid these inconsistencies, as will be discussed later.

In this study, we propose a generic parameterised patching procedure, see Table 15 for pseudo-code. Inputs to this procedure are summarised as follows:

TSP instance distance matrix $C = (c_{ij})$; a set of paths P to patch with cardinality $\#P$; the type of design of the patching operation, as specified by *type_of_patching_operation*, which depends on whether the chosen design is intended to expand the initial path or the initial subtour, where *type_of_patching_operation* = 0 refers to a design where the patching procedure patches all paths, a subset of paths at a time, to form a larger path Π, then connects the head and tail of the largest path Π to form a subtour, whereas *type_of_patching_operation* = 1 refers to a design where the patching procedure starts by merging a subset of paths to form a subtour S, then inserts the remaining paths into S. In this thesis, we only investigated the second option, e.g.

type_of_patching_operation = 1, thus, we only present the related details for this option.

When expanding the initial subtour, one could for example choose amongst several options such as insert a path into the subtour S being expanded by breaking an edge in S, break a path into subpaths and insert the subpaths into the subtour S being expanded at different places, and decision rule-based choice between insert and break and insert. The criterion according to which paths to merge are selected is referred to as *paths_to_merge_selection_criterion* and the criterion according to which paths to patch are selected as specified by *paths_to_patch_selection_criterion*; the measure of the performance criterion to optimise when performing the patching operation is called *patching_operation_performance_criterion*, and the type of implementation of the patching operation, as specified by *type_of_implementation*, where *type_of_implementation* = 0 refers to sequentially patching paths, whereas *type_of_implementation* = 1 refers to patching paths in parallel; the number of paths to merge or patch at a time n_{paths}^{patch} to obtain an initial subtour or path to be expanded; and the number of paths to patch at a time n_{paths}^{insert} in expanding the initial subtour or path. We propose *merging_criterion* that specifies how the chosen paths will be merged to construct a subtour, which could be either saving-based criterion or nearest merger criterion. The idea behind the *merging_criterion* is borrowed from basic construction heuristics, however, instead of nodes, we make use of paths. Consequently, they have differences; such as the construction of subtours and saving/merging calculations, where only the tail and head of the path is considered. The proposed *Nearest merger criterion* starts with subtours with a single-path and keeps merging pairs of subtours in an optimal manner until a single subtour is obtained, see Table 16. As for the proposed *saving -based criterion*, a merging cost matrix PMC is computed, see Table 17, where only the tail and head of the paths are considered in calculation of merging cost of each pair of paths (p_i, p_j). In addition, the merging considered is without crossover, see Figure 11.

Note that the subtour $S = \left\{\binom{p_1}{dir_{p_1}}, \binom{p_2}{dir_{p_2}}, \dots, \binom{p_1}{dir_{p_1}}\right\}$, obtained by the generic patching procedure, is a sequence of paths p_j, and their direction dir_{p_j}, where dir_{p_j}

takes either values of $\{1, -1\}$. If $dir_{p_j} = 1$ then p_j is not reversed in the subtour, $\overrightarrow{p_j}$; otherwise, $p_j$is reversed in the subtour, $\overleftarrow{p_j}$. In addition, choice of n_{paths}^{insert} should be less or equal to $|P| - n_{paths}^{patch}$, obviously if $n_{paths}^{patch} = |P|$, all paths will be merged and $n_{paths}^{insert} = 0$.

patching_method (C, *type_of_patching_operation, paths_to_merge_selection_criterion, paths_to_patch_selection_criterion, merging_criterion, patching_Criterion*, n_{paths}^{patch}, n_{paths}^{insert}, patching_operation_performance_criterion, type_of_implementation) **{**

Initialisation Step

Select the n_{paths}^{patch} paths to merge or patch according to *paths_to_merge_selection_criterion* or *paths_to_patch_selection_criterion* and its measure;

IF *type_of_patching_operation* = 1 **THEN** Merge the n_{paths}^{patch} paths into a subtour *S* according to the type of merging operation, the measure of the merging criterion, and the type of implementation chosen;

ELSE Patch the n_{paths}^{patch}paths into a larger path *Π* according to the type of patching operation, the measure of the patching criterion, and the type of implementation chosen;

Update *P* accordingly;

Iterative Step

WHILE $P \neq \emptyset$ **DO** {

Select n_{paths}^{insert} paths in *P* according to the selection criterion and expand the current subtour *S* or path *Π* according to the type of patching operation, the path patching criterion and its measure, and the type of implementation chosen;

Update the set of paths yet to be patched; i.e., delete the n_{paths}^{insert} paths selected above from *P*;

}

IF *type_of_patching_operation* = 0 **THEN** Connect the head and tail of the current path to form a subtour;

}

Table 15 Pseudo-code of the proposed generic patching procedure

The proposed *saving-based criterion*, Table 17, starts with the smallest path, p_0, as the center and initialises subtours as an optimal return from any other path to p_0; then merges a pair of subtours, at a time, with maximum saving until all subtours are merged.

Input:

$c = (c_{ij})$; n_{paths}^{patch}; and P : A set of paths to patch

Output:

A single subtour / tour, $S = \left\{\binom{p_1}{dir_{p_1}}, \binom{p_2}{dir_{p_2}}, \dots, \binom{p_1}{dir_{p_1}}\right\}$, see Figure 9

Initialisation steps

Create $|P|$ number of subtours with one path, $s_i = \left\{\binom{p_i}{1}, \binom{p_i}{1}\right\}$, see Figure 8 (a)

Initialise subtours distance matrix $SD(s_i, s_j)$.

Iterative steps

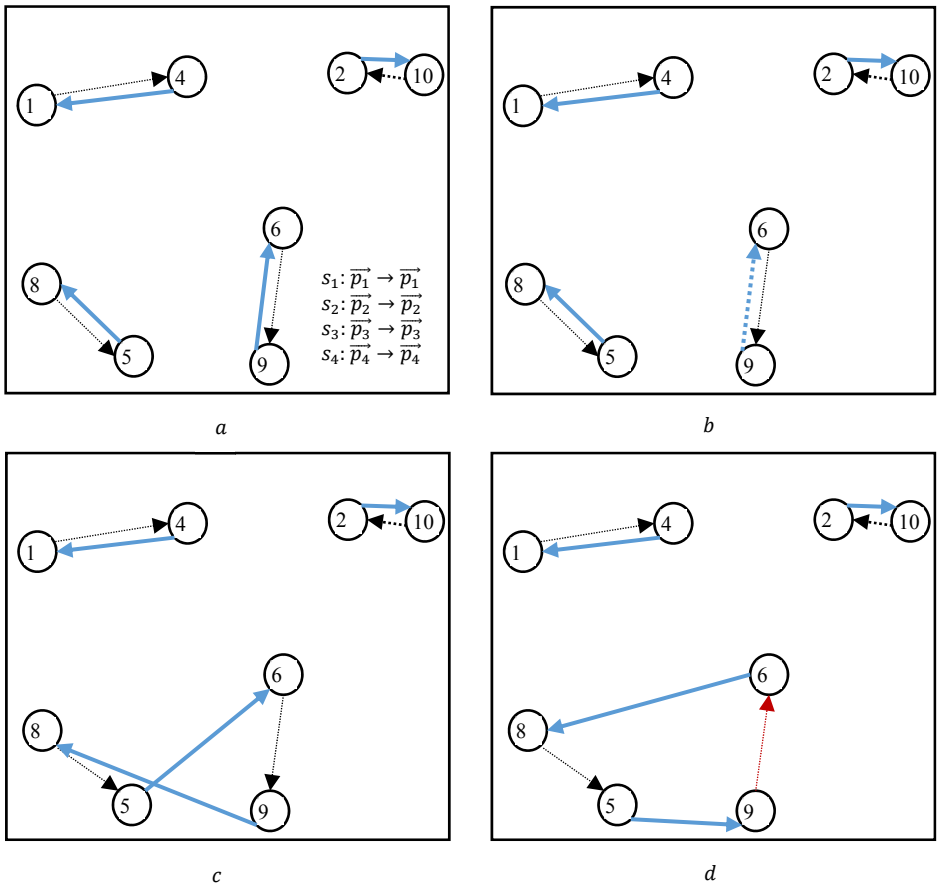
Repeat until all subtours are merged

Find the two closest subtours (s_i, s_j), see Figure 6

Merge them in the best possible way, without crossover (Figure 8, c), and reverse paths if necessary (Figure 8, d), and update SD.

End Repeat

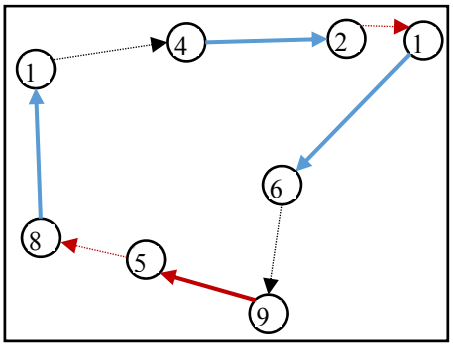
Table 16 Nearest merger method



c) $Merging(\overrightarrow{p_2} \rightarrow \overrightarrow{p_4} \rightarrow \overrightarrow{p_2}) = c(5,6) + c(9,8) - c(5,8) - c(9,6)$

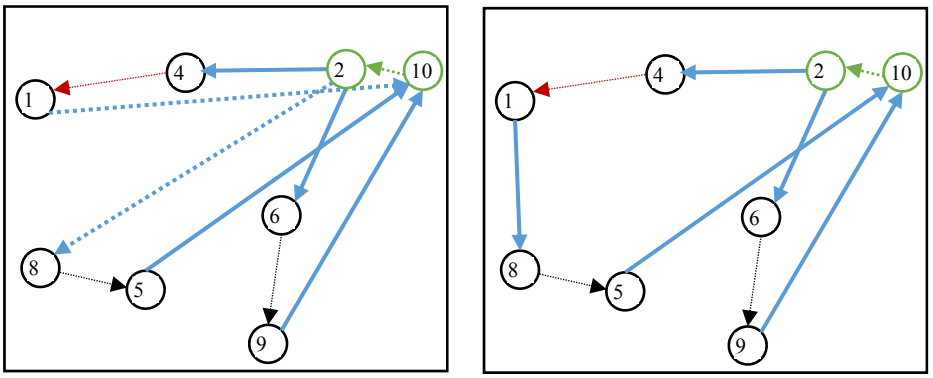
d) $Merging(\overrightarrow{p_2} \rightarrow \overleftarrow{p_4} \rightarrow \overrightarrow{p_2}) = c(5,9) + c(6,8) - c(5,8) - c(9,6)$

Figure 8 Nearest merger criterion, merging process



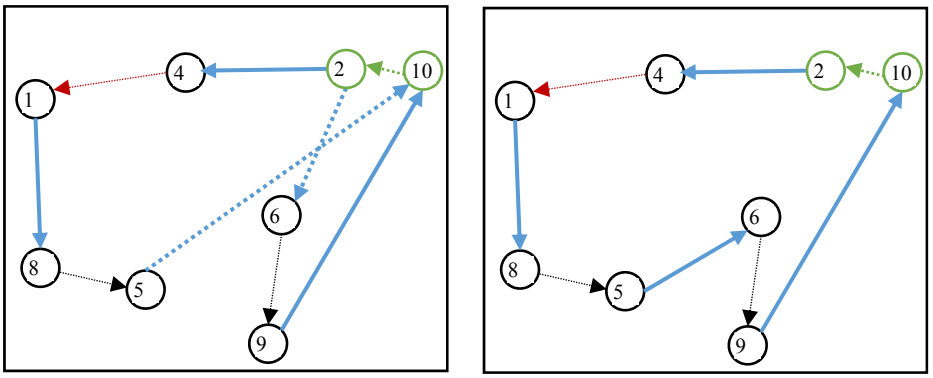
$$S = \left\{\binom{p_1}{1}, \binom{p_3}{-1}, \binom{p_4}{1}, \binom{p_2}{-1}, \binom{p_1}{1}\right\}$$

Figure 9 Nearest merger criterion, final solution



a b

$$Saving(s_1, s_2) = \text{Cost}(\overrightarrow{p_1} \rightarrow \overrightarrow{p_0}) + \text{Cost}(\overrightarrow{p_0} \rightarrow \overrightarrow{p_2}) - \text{Cost}(\overrightarrow{p_1} \rightarrow \overrightarrow{p_2})$$
$$= c(1,10) + c(2,8) - c(1,8)$$



c d

$$Saving(s_1, s_2) = c(5,10) + c(2,6) - c(5,6)$$
$$S = \left\{\binom{p_0}{1}, \binom{p_1}{-1}, \binom{p_2}{1}, \binom{p_3}{1}, \binom{p_0}{1}\right\}$$

Figure 10 Saving-based patching, Initialising subtours

Input

$c = (c_{ij})$; n_{paths}^{patch}; P

Output

A single subtour / tour, S

Initialisation steps

Choose the path p_0 with smallest cardinality.

For each remaining path p_i {

Construct a subtour consisting p_0 and p_i, $s_i = \left\{\binom{p_0}{1}, \binom{p_i}{dir_{p_i}}, \binom{p_0}{1}\right\}$, without crossover and reverse path $\overrightarrow{p_i}$ if necessary ($dir_{p_i} = -1$), see Figure 10;

For each pair of subtours (s_l, s_k) calculate the least savings obtained by patching them, see Figure 11, and sort the savings in a non-increasing order;**}**

Iterative steps

Repeat until all subtours are patched

Patch the two subtours with the maximum savings;

IF all the subtours are patched **Then** stop;

Else update the savings list and go to step 1;

End Repeat

Table 17 Saving based path patching method

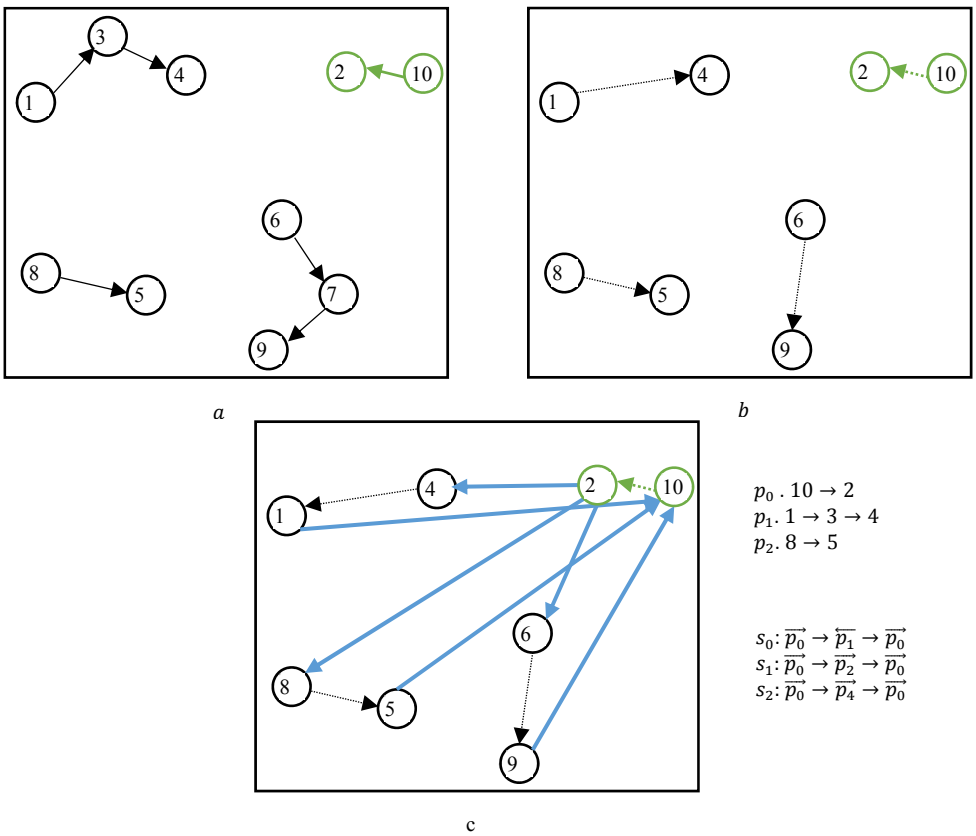


Figure 11 Savings calculation

The proposed *insertion_criterion* defines how to insert the chosen paths in the subtour in the best possible location, which is based on the cheapest insertion cost of paths in the subtour, Figure 12.

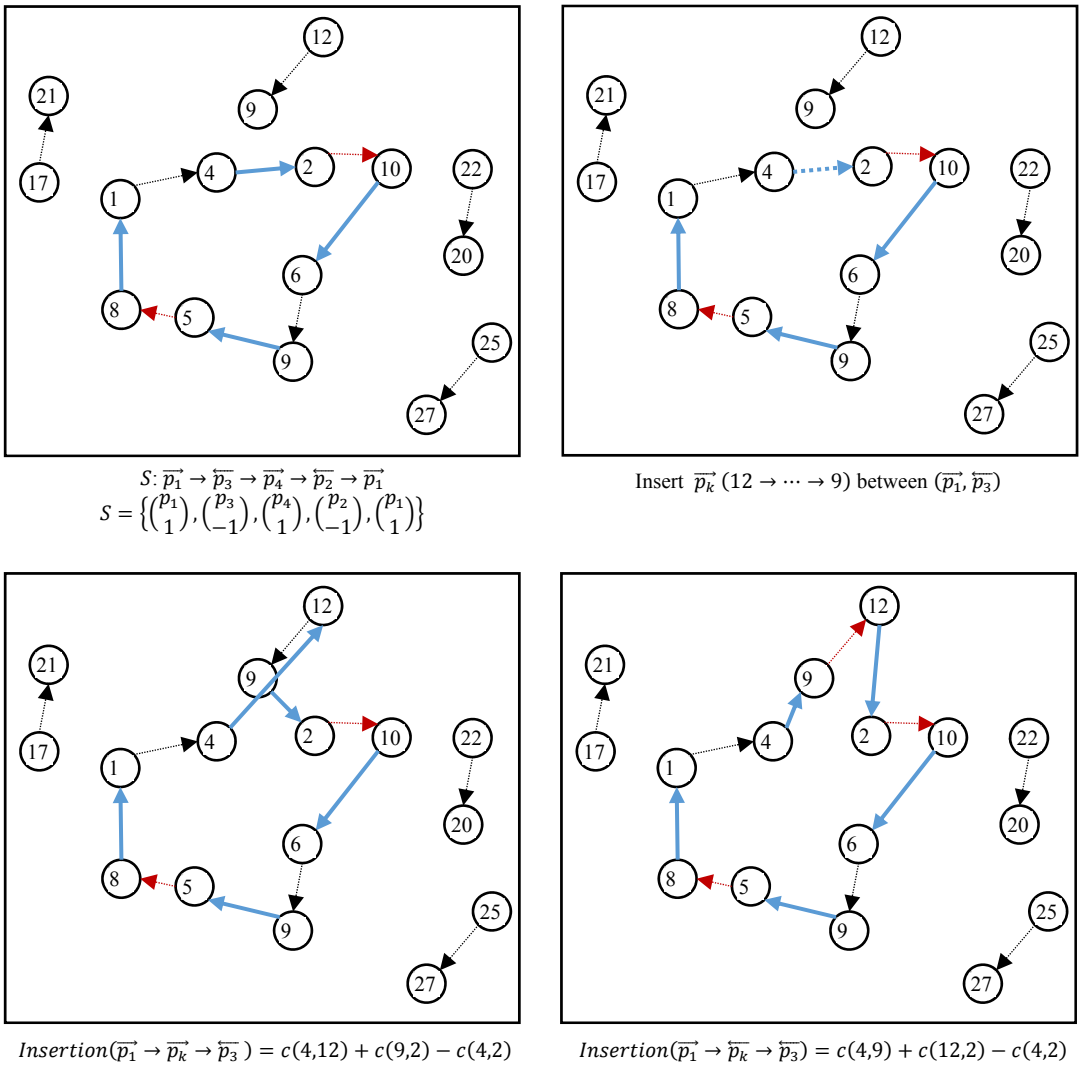


Figure 12 Cheapest insertion

The $paths\_to\_merge\_selection\_criterion$, Table 18, can be categorised as cardinality-based criterion (largest or smallest), cost-based criterion (longest or shortest), distance-based criterion (closest or farthest) and merger cost-based criterion (cheapest or expensive). The default category could be arbitrary. Note that, the distance-based criterion makes use of paths distance matrix,PD, which is obtained by the algorithm in Table 19, where the distance between each pair of paths (p_k, p_ℓ) is the minimum distance between all nodes in path p_k and all the nodes in path p_ℓ.

Paths to merge Selection Criterion	Description
Arbitrary	Random choice of paths
Cardinality-Based	Based on number of nodes in the path
Cost-Based	Based on travel cost, length of the path
Distance-Based	Based on the distance between the paths
Merging Cost-Based	Based on the cost of merging each pair of paths

Table 18 Paths to merge selection criterion

Input

$c = (c_{ij})$; P : A set of paths to patch with cardinality $|P|$

Output

$PD(p_k, p_\ell)$: Paths distance matrix

Initialisation Step

Initiate paths distance matrix $PD(p_k, p_\ell) = \infty$;

Iterative Step

For each pair of paths (p_k, p_ℓ) compute the minimum distance between path p_k and path p_ℓ as follows:

$PD(p_k, p_\ell) =$

$min\{c(p_k(1), p_\ell(1)), c(p_k(1), p_\ell(\#p_\ell)), c(p_k(\#p_k), p_\ell(1)), c(p_k(\#p_k), p_\ell(\#p_\ell))\};\}$

Table 19 Paths distance matrix

Input:

$c = (c_{ij})$; P

Output:

$PMC(p_i, p_j)$: Merging cost matrix

Initialisation Step

Initiate $PMC(p_i, p_j) = \infty$;

Iterative Step

FOR each pair of paths (p_i, p_j) compute their merging cost where a path $p \in P$ is specified by its cardinality $\#p$ and its sequence of nodes $(p(1), \ldots, p(\#p))$, as follows:

$Merging\_Cost(\overrightarrow{p_i} \rightarrow \overrightarrow{p_j} \rightarrow \overrightarrow{p_i}) = \left(p_i(\#p_i), p_j(1)\right) + c\left(p_j(\#p_j), p_i(1)\right)$;

$Merging\_Cost(\overrightarrow{p_i} \rightarrow \overleftarrow{p_j} \rightarrow \overrightarrow{p_i}) = \left(p_i(\#p_i), p_j(\#p_j)\right) + c\left(p_j(1), p_i(1)\right)$;

IF $(Merging\_Cost(\overrightarrow{p_i} \rightarrow \overrightarrow{p_j} \rightarrow \overrightarrow{p_i}) \leq Merging\_Cost(\overrightarrow{p_i} \rightarrow \overleftarrow{p_j} \rightarrow \overrightarrow{p_i}))$ **THEN**

$PMC(p_i, p_j) = Merging\_Cost(\overrightarrow{p_i} \rightarrow \overrightarrow{p_j} \rightarrow \overrightarrow{p_i})$;

Else

$PMC(p_i, p_j) = Merging\_Cost(\overrightarrow{p_i} \rightarrow \overleftarrow{p_j} \rightarrow \overrightarrow{p_i})$;

Table 20 Paths merging cost matrix

As for the proposed merging cost-based criterion, a merging cost matrix PMC is computed, see Table 20, where only the tail and head of the paths are considered in calculation of merging cost of each pair of paths (p_i, p_j). In addition, the merging considered is without crossover, see Figure 8 (c).

In the iterative step, we categorised *paths_to_patch_selection_criterion* into cardinality-based criterion (largest or smallest), cost-based criterion (longest or shortest), distance-based criterion (closest or farthest) and insertion cost-based criterion (cheapest or expensive), see Table 21.

The proposed cardinality-based and the cost-based criterion is similar to the ones for *paths_to_merge_selection_criterion.* However, the proposed distance-based criterion is based on the distance between the paths in the subtour and the remaining paths not in the subtour, PSD, Table 22. On the other hand, the proposed insertion cost-based criterion is based on the cost of inserting the paths not in the subtour in the subtour, without crossover, see Table 23.

Paths to Patch Selection Criterion	**Description**
Arbitrary	Random choice of paths
Cardinality-Based	Based on number of nodes in the path
Cost-Based	Based on travel cost, length of the path
Distance-Based	Based on the distance between the remaining paths and the subtour
Insertion Cost-Based	Based on the cost of inserting each of the remaining paths in the subtour

Table 21 Paths to patch selection criterion

Input:

$c = (c_{ij})$; P

Output:

$PSD(p_k, p_l)$: Path to subtour distance matrix

Initialisation Step

Initiate $PSD(p_k, p_l) = \infty$;

Iterative Step

For each path p_k not in the subtour compute the minimum distance between each path p_k and all the paths, p_l ,in the subtour S as follows:

$PSD(p_k) = \min\{PD(p_k, p_\ell) \quad l = 1, \dots, |S|\}$; //Where S consists of $|S|$ number of paths

Table 22 Distance-based criterion

Input:

$c = (c_{ij})$; P

Output:

$PCIS(p_k, p_l)$: Path cheapest insertion cost into subtour matrix

Initialisation Step

Initiate $PCIS(p_k, p_l) = \infty$;

Iterative Step

For each path p_k not in the subtour calculate the cheapest insertion cost of p_k between each pair of paths in the subtour (p_l, p_{l+1}), without crossover and reverse paths if necessary, see Figure 12.

Table 23 Cheapest insertion

3.2.6. Performance metric

The specification of the performance metric to be used for assessing infeasible neighbours or equivalently the metric to use for comparing the performance of the break and patch or repair operations is represented by a categorical variable; namely, *IMetric*. Several categories could be considered to reflect different aspects of the repair mechanism and its "quality". For example, one could consider the cost of the solution after repair to compare different repair operations, in our empirical investigation; we considered this metric as the default choice. Alternatively, one could consider the number of arcs, that cross-over, which is to be minimised. Also, one could define categories that are concerned with optimizing more than one metric; for example, one could minimise both the cost of the solution after repair and the number of arcs in the solution that crossover.

3.2.7. Improvement mechanism

With respect to the local improvement mechanism of components of the infeasible neighbours generated by Type I moves, or equivalently the Type II moves, this decision is represented by a categorical variable *T2M*. A variety of categories could be defined; e.g., 2-opt, 3-opt and combinations of these moves. These moves are implemented by means of an improvement mechanism represented by a categorical variable *component_improvement_mechanism*. The categories of this variable would correspond to different local search methods including metaheuristics. As to the performance metric to be used for assessing different components, this choice is made

through a categorical variable *component_performance_metric*. The default category would be the commonly used metric; namely, the cost or total distance of the component. Other categories could be defined by the analyst to consider additional features of a component or to take account of soft or hard constraints not explicitly considered in the problem formulation. Besides, one has to decide how often to call upon the local improvement mechanism of components of the infeasible neighbours generated by Type I moves; namely, improvement at each iteration, deterministic static and stochastic.

3.3. Infeasible neighbourhood structure

Finally, we have to choose the design of the infeasible-based neighbourhood structure (*INS*) to use in exploring the infeasible space of the TSP. One of two alternatives could be chosen. The first option is to use Type I moves to explore the infeasible space and select the best neighbour, then Type II moves are used to improve the best neighbour locally – we refer to this design as improving best neighbour (IBN) design. The second option is to immediately use Type II moves to improve every neighbour obtained with Type I moves – we refer to this design as improve all neighbours (IAN) design.

3.4. Implementation of GPILS

The implementation of the proposed GPILS could prove rather tricky without a proper algorithm to carry the steps required for exploring the infeasible space for any specification of the parameters of the search. Note that, by set of parameters we mean set of parameters, criteria, components and rules but for simplicity this set is referred to as set of parameters. In fact, exploring the infeasible space for a given choice of the parameters of the search, say (s,r), requires exploring all, or sometimes most, combinations of r arcs in s subtours which in turn requires a minimum of s embedded FOR loops. Obviously, the number of embedded FOR loops required from one run or experiment to another is different and requires a different code. In order to make the code generic;

RINS

$\left(s^k, \{S_i^k\}, \{r_i^k\}, \{CS_i^k\}, \left\{e_{S_i^k}\right\}, m, \ell, j_m^\ell, C, DNS, T1M, T2M, IMetric, \bar{\bar{y}}_k^*, IMetric(\bar{\bar{y}}_k^*), patching\_method\right)$ **{**

// If r_i^k edges to break were identified in all subtours S_i^k, then implement the type I move;

// else, identify the edges to break in the remaining subtours to break

IF $(m > s^k)$ **{**

// Break and patch the subtours to merge using type I move to obtain a single subtour $\bar{\bar{y}}_k$

$\bar{\bar{y}}_k$ =**PerformTypeIMove**$\left(s^k, \{S_i^k\}, \{r_i^k\}, \left\{e_{S_i^k}\right\}, C, patching\_method\right)$;

// Improve the current infeasible solution locally using Type II moves, if required

IF INS = IAN **THEN** $\bar{\bar{y}}_k$ = **PerformTypeIIMove**$(T2M, \bar{\bar{y}}_k)$;

// Update the best neighbour and its cost, if necessary

IF ($IMetric(\bar{\bar{y}}_k)$ is better than $IMetric(\bar{\bar{y}}_k^*)$ and within the bounds) **THEN { ** $\bar{\bar{y}}_k^* = \bar{\bar{y}}_k$; $IMetric(\bar{\bar{y}}_k^*) = IMetric(\bar{\bar{y}}_k)$; **}}**

ELSE {

// Identify the edges to break in subtour m for which such edges are yet to be specified. Note

// that the FOR-loop condition $|S_m^k| - (r_m^k - \ell)$ is chosen so that any repetition of combinations

// of edges is avoided

FOR $j_m = j_m^\ell \; to \; |CS_m^k| - (r_m^k - \ell)$ **{**

// Consider the jth edge in candidate set CS_m^k of edges to break in subtour S_m^k

Set $e_{S_m^k}(\ell) = CS_m^k(j_m)$;

$\ell = \ell + 1$; // Increment edge index counter ℓ to loop through all remaining edges in subtour m

IF $(\ell \leq r_m^k)$ **THEN** $j_m^\ell = j_m + 1$; // Set the loop ℓ initialiser j_m^ℓ to $j_m + 1$

ELSE {

// Increment subtour index counter m to loop through all remaining subtours or embedded

// FOR loops

$m = m + 1$;

$\ell = 1$; // Reset edge index counter ℓ to 1

$j_m^\ell = 1$; // Reset loop ℓ initialiser of edge index j_m in subtour m to 1 **}**

RINS$(s^k, \{S_i^k\}, \{r_i^k\}, \{CS_i^k\}, \left\{e_{S_i^k}\right\}, m, \ell, j_m^\ell, C, DNS, T1M, T2M, IMetric, \bar{\bar{y}}_k^*, IMetric(\bar{\bar{y}}_k^*), patching\_method)$;

$\ell = \ell - 1$; // Decrease edge index counter ℓ to go back to the previous FOR loop

IF $(\ell = 0)$ **{**

$m = m - 1$; // Decrease subtour index counter m to go back to the previous FOR loop

$\ell = r_m^k$; // Reset edge index counter ℓ to go back to the previous FOR loop

}

}}}}

Table 24 Pseudo-code of the RINS function

i.e., could be run with any specification of the search parameters, we propose a recursive infeasible neighbourhood search function (RINS), Appendix I. For a detailed pseudo-code see Table 24.

3.5. Choice of how to explore the primal space

Exploring the primal space is optional in our design of GPILS. When the option of exploring the primal space is chosen, one has to choose the primal neighbourhood structure to use, the improvement mechanism as well as the performance metric to be used for assessing primal neighbours. The choice of whether to explore the primal space or not is represented by a categorical variable *explore_primal_space*. This variable is set to 0 when this option is turned off, 1 when exploring the primal space is considered using a primal improvement mechanism, and 2 when exploring the feasible space while allowing for infeasibilities; that is, using a feasible-infeasible mechanism. When this option is on, one could choose from a variety of primal neighbourhood structures as specified by a categorical variable *primal_neighbourhood_structure*, where categories would correspond to 2-opt moves, 3-opt moves, or combinations of these basic moves. The improvement mechanism is represented by a categorical variable *primal_improvement_mechanism*, where the categories would relate to different local search methods including metaheuristics.

As to the performance metric to be used for assessing primal neighbours, this choice is made through a categorical variable *PMetric*. The default category would be the commonly used metric; namely, the cost of the primal solution. Other categories could be defined by the analyst to consider additional features of a primal solution; e.g., the minimum pollution, total travel time and etc. Note that through the choice of the primal performance metric one could address in an infeasible fashion additional constraint such as time windows constraints. The proposed GPILS is by design a parameterised solution framework. In the next section, we propose a hyperheuristic framework for its implementation. The aim of this hyperheuristic framework is to optimise the choice of the parameters of GPILS by exploring the corresponding parameter space using a local search framework.

3.6. DLS versus GPILS

Conceptually, the proposed GPILS and DLS developed by Ouenniche et al. (2017) are similar, although their implementation is different. We refined and enhanced their design and pushed it forward to the settings that has not been considered by DLS. A summary of the comparative analysis between DLS and GPILS is shown in Table 25. The refinements are as follows:

Initialisation of the lower bound and the seed: as compared to DLS, where only AP-relaxation is considered to initialise the bound and the seed, GPILS also considers *PIH*.

Parameters and operations	**DLS**	**GPILS**
Initialisation of the upper bound	Nearest merger	Construction heuristics
Initialisation of the dual bound and the seed	*AP* –	*AP* *PIH*, see Table 11
Number of subtours to break and patch	2,3	$2, \dots, N_{sbt}$
Subtours selection criteria	Farthest/ nearest distance between subtours; cheapest cost of merger of subtours	See Table 12
Number of arcs involved in repair mechanism	$1, \{2,1\}$	$1, .., \lvert P \rvert$
Arcs selection criteria	All combinations; –	All combinations; K-NN
Patching operation	All combinations	Generic patching procedure; see Table 15
Infeasible neighbourhood structure	IBN	IBN
Improvement mechanism	Local search; 2-opt, 3-opt and US moves; Improvement at each iteration, deterministic and stochastic; –	Local search; 2-opt, 3-opt; Improvement at each; Reinforcement
Implementation	Static implementation	Generic implementation; see Table 24
Exploring the primal space	None	None

Table 25 Comparative analysis between DLS and GPILS

Number of subtours s and arcs r involved in repair mechanism: in comparison with DLS, GPILS can consider wider range of values. However, for computational time we considered upper bound for both, i.e. $s = 2, \dots, 5$ and $r = 1, \dots, 5$.

Subtours selection criteria: GPILS considers wider range of criterion, see Table 12.

Arcs selection criteria: DLS is a greedy method where it considers all combination of breaking arc in each subtour. On the other hand, GPILS considers a restricted selection criterion where the arcs to break are chosen amongst candidate sets obtained by *K-NN*.

Patching operation: yet again, DLS uses a greedy method where it considers all combinations of patching the broken subtours. As for GPILS, a generic patching procedure is used to patch the broken subtours, for more detail see Table 15.

Improvement mechanism: both DLS and GPILS consider local search as an improvement mechanism to locally improve the intermediate infeasible solutions. As for *T2M*, DLS considered 2-opt, 3-opt and US moves. However, because the US move is computationally inefficient, it is not considered in GPILS. Regarding how often the local improvement mechanism is been called DLS considered improvement at each iteration, deterministic static and stochastic, however, only improvement at each iteration is used in GPILS since it was the best performing option.

Implementation: DLS is static implementation design, meaning that for any combination of (s, r) the number of embedded FOR loops required from one run or experiment to another is different and requires a different code. On the other hand, GPILS is a generic design that could run with any combination of (s, r).

3.7. Empirical results

In this section, we shall compare the proposed GPILS under different settings. In order to see the performance of GPILS under different settings for different instances, we made a step by step experiment. In the first step of the experiment, we chose the value of the parameters similar to DLS, so we could compare GPILS with DLS. Note that the GPILS is conceptually like DLS, however, their implementation is different. Thus, the difference in quality of the solution is expected. The aim of this experiment is to

empirically demonstrate that GPILS performs better than DLS on both quality of the solution delivered and computational time. In the next steps of the experiments, we push forward the analysis by considering further setups that has not been considered by DLS which our framework allows for. To do so, in each step of the experiment, we fixed all the parameters except one or two and we summarised the performance of GPILS. For more details of this experiment refer to Table 27.

3.7.1. Experimental setup

All methods are implemented in C# and tested on a Windows 7 Enterprise with 2.26 GHz Core i5 processor and 16 GB of RAM. The AP-based relaxation is solved using CPLEX 12.5. The empirical results are based on problem instances from TSPLIB. The problem instances are presented in Table 26.

Nodes	51	76	76	100	100	100	100	100	101	107	124	130	136	144	150	150	150	152	200	200
Name	eil51	eil76	pr76	kroA100	kroB100	kroC100	kroD100	kroE100	eil101	pr107	pr124	ch130	pr136	pr144	ch150	kroA150	kroB150	pr152	kroA200	kroB200

Table 26 Problem instances

For the *choice of the parameters of GPILS* in this empirical investigation, we experimented with the following parameters:

Parameters of the bounding scheme

- PM: Arbitrary insertion; nearest insertion; farthest insertion; cheapest insertion; Clarke and Wright; nearest merger
- *IM* : {AP, *PIH* }
- Parameters of *PIH*
  - N_{Sbt}: $1, \dots, 20$
  - $CO = 1$
  - DRA: $\mathcal{K}$-means ($\mathcal{K} = N_{Sbt}$)
  - DRC: Construction heuristic similar to PM
  - Imp:{0, 1}

- IM: Classic local search
- NS: 2-opt; 3-opt;

Parameters of Type I move

- Breaking operation
  - s: $2, \dots, 5$
  - $subtours\_selection\_criterion$: Random; shortest/ largest subtours; smallest/ largest subtours; closest/ farthest subtours; cheapest/ most expensive cost of merging pair of subtours
  - r: $1, \dots, 5$ (if $r_i^k > |S|$ then $r_i^k = |S| - 1$)
  - $arcs\_to\_break\_selection\_criterion$: K-NN ($K = 1, \dots, 10$)
- Patching operation
  - *type_of_patching_operation* = 1
  - Initialisation step
    - $\boldsymbol{n_{paths}^{patch}}$: $1, \dots, P$
    - $paths\_to\_merge\_selection\_criterion$: Largest / smallest; longest/ shortest; closest/ farthest; cheapest/expensive merging cost
    - $merging\_criterion$: *Saving-based path merging (SPM); nearest path merger (NPM)*
    - $type\_of\_implementation = 0$
  - Iterative patching
    - $\boldsymbol{n_{paths}^{insert}}$:[0,1]
    - $paths\_to\_patch\_selection\_criterion$: Largest / smallest; longest/ shortest; closest/ farthest;
    - $patching\_criterion$: Cheapest Insertion
    - $type\_of\_implementation = 0$
  - $patching\_operation\_performance\_criterion$: Cost of the subtour

Parameters of Type II move

- *T2M*: 2-opt; 3-opt;
- $component\_improvement\_mechanism$: Local search
- $component\_performance\_metric$: Cost or total distance of the component

Note that, we also experimented 'reinforced improvement'. In other words, after improving the degree of infeasibility of the chosen *T2M*, we improved the solution using a different *T2M,* again.

Other parameters of GPILS

- $IMetric:$ Cost or total distance of the component
- INS: IBN

Parameters of the primal space exploration

- $explore\_primal\_space = 0$

In order to understand the effect of different sets of parameters of GPILS, we experimented with GPILS given several sets of parameters. Note, however, that for space constraints, we only present a number of these sets.

3.7.2. Experimental results

In this section, we shall compare the proposed GPILS under different settings. In order to see the performance of GPILS under different settings for different instances, we made a step by step experiment. For more details of this experiment refer to Table 27. In the first experiment, we chose the value of the parameters similar to DLS, to some extent, so we could compare GPILS with DLS. Thus, we set the parameters of the first experiment as follows:

Parameters of the bounding scheme

- PM:∞;
- *IM*: AP
- Parameters of *PIH*: since *IM* is set to AP, these parameters are not required

Parameters of Type I move

- Breaking operation
  - s: $2, 3$
  - $subtours\_selection\_criterion$: farthest distance between subtours
  - r: $1,\ 2$
  - $arcs\_to\_break\_selection\_criterion$: K-NN ($K = 1, \dots, 10$)

- Patching operation
  - Initialisation step
    - n_{paths}^{patch}: P
    - $paths\_to\_merge\_selection\_criterion$: since n_{paths}^{patch} is set to P, this parameter is not required
    - $merging\_criterion$: nearest merger
- Iterative patching: since n_{paths}^{patch} is set to P, this parameter is not required

Parameters of Type II move

- *T2M*: 2-opt; 3-opt;

The rest of the parameters are set to the values mentioned before. As for the 'reinforced improvement', it is not used in the first experiment. Note that before comparing GPILS with DLS, we experiment with$arcs_to_break_selection_criterion$; namely, *K-NN* with *K* ranging from one to ten, in order to reduce the computational time. Later, in the next experiments, we fixed the value all the parameters except for one or two of them, so we could see the quality of GPILS under different settings for different instances with different structures.

Not that the conclusions made in this section could be different, even opposite, for different settings of parameters. Hereafter we shall present these experiments in more detail as well as their result. Bear in mind that the statistics presented in this section are the average percentage increase over the optimal solution (i.e. $performance = \frac{tour\ cost - known\ optimum}{known\ optimum} \times 100\%$). Moreover, in the following figures each bar represent different values for the specified parameter(s) in each experiment and positive values means that the GPILS given new settings perform better than the previous experiment, unless otherwise is noted.

I. Experiment 1: K-NN

The proposed GPILS is a parameterised neighbourhood structure, which allow us to intensify and diversify the search, depending on the chosen parameters and structure of the problem, while controlling the rate of convergence of the process toward the feasible solution.

7: *PIH vs AP*	**6:** $\boldsymbol{p}aths\_to\_patch\_selection\_criterion$	**5:** $\boldsymbol{p}aths\_to\_merge\_selection\_criterion$	**4:** $(n_{paths}^{patch}, merging\_criterion)$	**3:** $subtours\_selection\_criterion$	**2:** (s, r)	**1: K-NN**	**Experiment**
_	_	_	_	_	_	_	***PM***
PIH	AP	AP	AP	AP	AP	AP	***IM***
2:2:20	_	_	_	_	_	_	N_{sbt}
C&W Savings	_	_	_	_	_	_	***DRC***
1	_	_	_	_	_	_	***Imp***
3-opt	_	_	_	_	_	_	***NS***
3	3	3	3	3	2, … ,5	2,3	**s**
longest	longest	longest	longest	See Table	farthest	farthest	$\boldsymbol{subtours\_selection\_criterion}$
5	5	5	5	5	1, … ,5	1, 2	$\boldsymbol{r}$
r	r	r	r	r	r	$\mathbf{1}, \ldots, \mathbf{10}$	$\boldsymbol{arcs\_to\_break\_selection\_criterion}$
P	6	6	1, …,10	P	P	$\boldsymbol{P}$	$\boldsymbol{n_{paths}^{patch}}$
cheapest	cheapest	See Table	nearest	_	_	_	$\boldsymbol{paths\_to\_merge\_selection\_criterion}$
SPM	SPM	SPM	NPM-SPM	NPM	NPM	NPM	$\boldsymbol{merging\_criterion}$
_	1	1	1	0	0	0	$\boldsymbol{n_{paths}^{insert}}$
_	See Table	nearest	nearest	_	_	_	$\boldsymbol{paths\_to\_patch\_selection\_criterion}$
3-opt	3-opt	3-opt	3-opt	3-opt	3-opt	3-opt	**T2M**
_	_	_	_	_	_	_	***Reinforcement***

8 PIH-DRC	–	PIH	3	*DRC*	0	–	3	longest	5	***r***	P	cheapest	SPM	–	–	3-opt	–
9: PIH-NS	–	PIH	3	Farthest insertion	1	*NS*	3	longest	5	*r*	P	cheapest	SPM	–	–	3-opt	–
10: T2M & reinforcement	–	PIH	3	Farthest insertion	0	–	3	longest	5	*r*	P	cheapest	SPM	–	–	***T2M***	***reinforcement***
11:PM	PM	PIH	3	Farthest insertion	0	–	3	longest	5	*r*	P	cheapest	SPM	–	–	3-opt	–

Table 27 Experimental design

As it was mentioned earlier in this chapter, GPILS start with an initial infeasible solution, i.e. number of subtours, and progress towards the primal space by iteratively breaking a number of subtours, given (s, r), and patching them, until it lands in the feasible solution. In other words, low values for s and high values for r will lead to more diverse search and slower convergence rate, on the other hand, high values for s and low values for r will lead to less diverse search and faster convergence rate. A small illustrative example is shown in Figure 13.

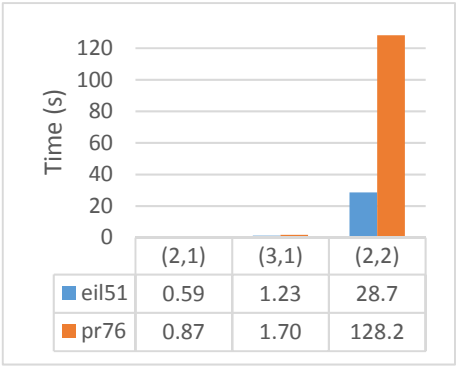


Figure 13 Computational time given (s, r)

Moreover, an initial infeasible solution with large number of subtours could also lead to higher diversity and lower convergence of the search. Thus, one should make a trade-off between diversity and convergence rate – see experiment 7.

Although, GPILS conceptually is similar to DLS, their implementation is different. Hereafter, we shall compare them empirically. As it was mentioned earlier, DLS developed by Ouenniche et al. (2017) required several parameters. Thus, in order to compare DLS with GPILS, we only considered the following parameters:

1. The choice of s subtours to merge: farthest distance between subtours, since it was best performing criteria.
2. The choice of number of subtours (s) to break and number of edges (r) to break in each subtour: $\{s = 2; r_1 = 1, r_2 = 1\}$ and $\{s = 3; r_i = 1, i = 1,2,3\}$
3. The choice of type II move: 3-opt.

   US move is not considered since it is computationally inefficient, although overall, it was the best criteria. On the other hand, 2-opt leads to good quality solution in much shorter time, not as good as 3-opt.

4. The choice of local improvement scheme: improvement in each iteration, since it was best performing criteria.

The comparison between GPILS and DLS is shown in Figure 14 and Figure 15. Each bar in these figures represent GPILS given the parameters in experiment 1 and K in range between one and 10. Note that positive values means that GPILS outperforms DLS and the average shows the difference between average $performance$ between DLS and GPILS, i.e. $\overline{performance}(GPILS) - \overline{performance}(DLS)$, where the average overall instances is shown as $\overline{performance}$. $K = 3$ and $K = 4$ for both experiments (2,1) and (3,1), respectively, outperforms DLS by %0.96 and %1.34, respectively. As you can see time increase when K increases, except for some cases, which could be because of structure of the problem instance. Thus, one has to make a trade-off between solution quality and computational time. Moreover, $K = 1$ or more specifically $K = r$ for both experiments, i.e. (2,1) and (3,1), outperforms DLS by 0.63 and 0.91, respectively. Consequently, in trade-off between solution quality and computational time, we fix $K = r$ in next experiments.

II. Experiment 2: (s, r)

In the second experiment, we present the effect of different values for s and r, shown by (s, r) and compared with s and r equal to $(2,1)$. This comparison is presented in

Figure 16, where each bar presents the comparison between each set of values of (s, r) and with s and r equal to $(2,1)$; i.e. $\overline{performance}(s, r) - \overline{performance}(2,1)$.

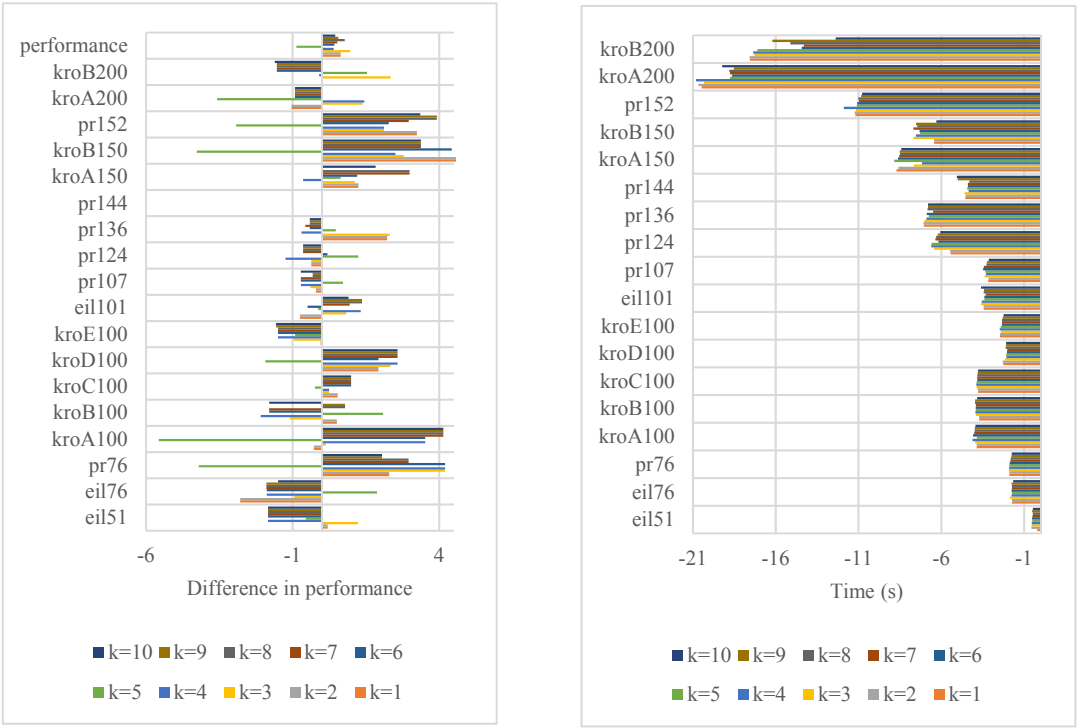


Figure 14 GPILS vs DLS given $s = 2$ and $r = 1$

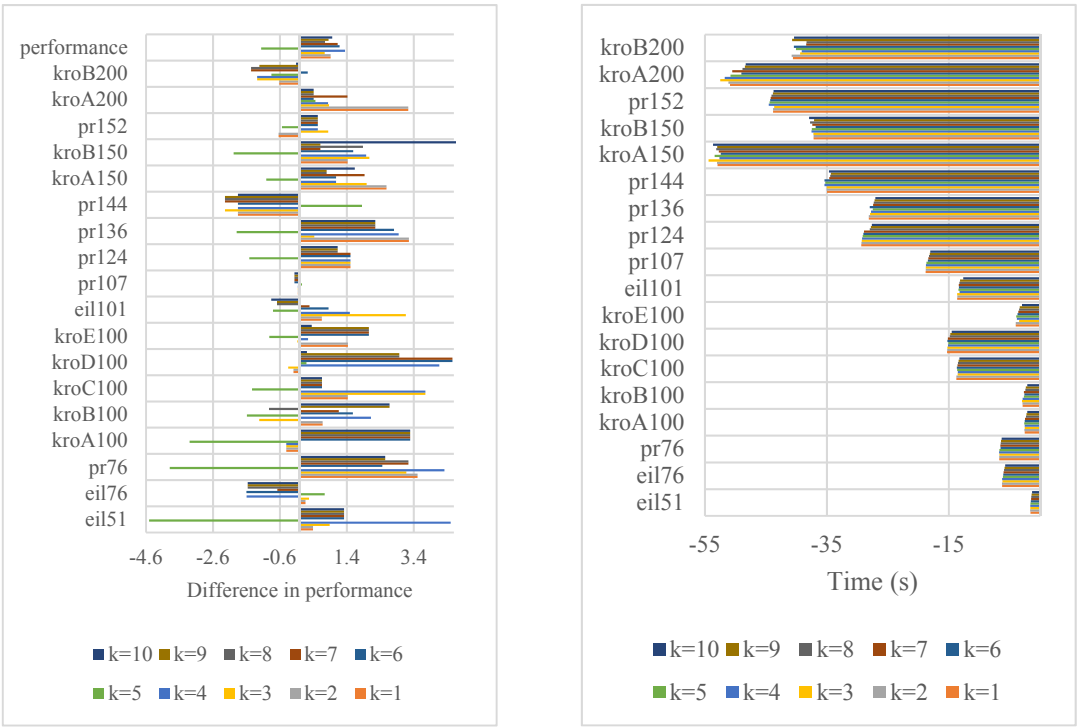


Figure 15 GPILS vs DLS given $s = 3$ and $r = 1$

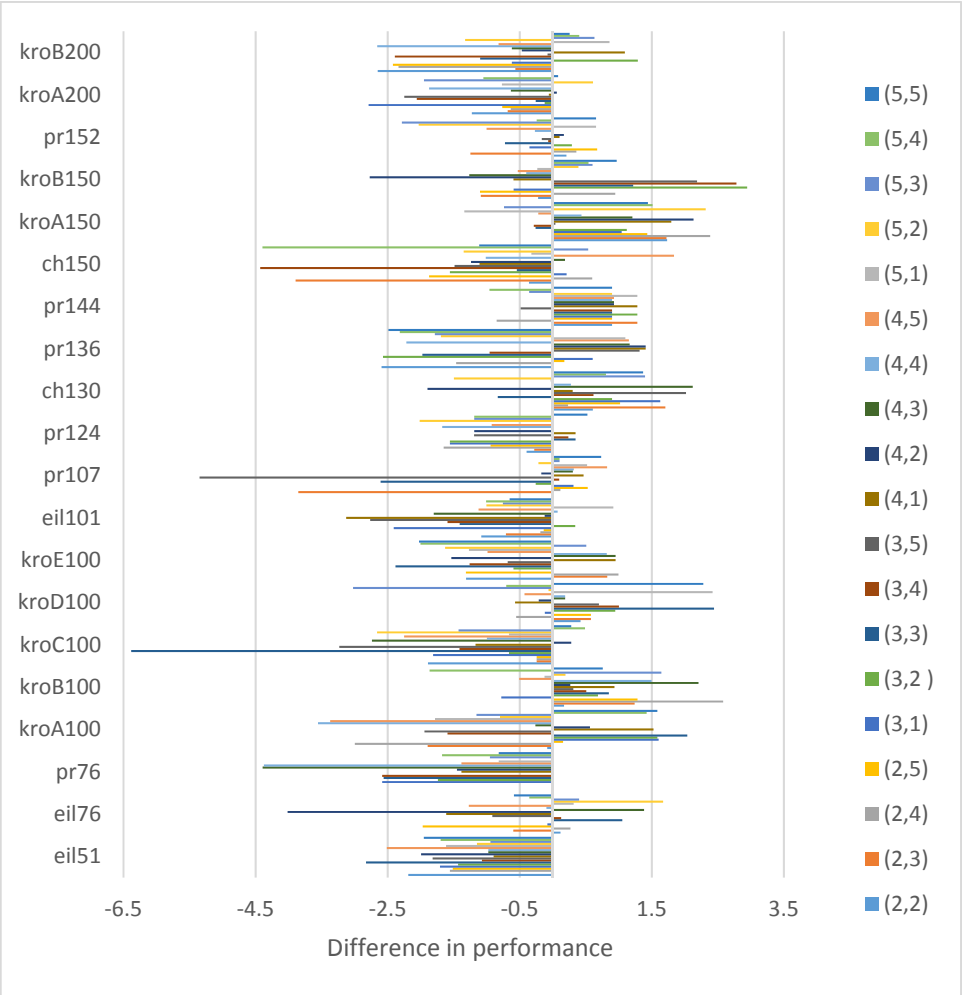


Figure 16 Performance of GPILS with set (s, r) *in comparison with* $(2,1)$

Note that, positive values mean that when s is set to two and r is set to one, the GPILS produces worse solutions than GPILS given other sets.

Overall, on average the best performing sets of values for s and r are (3,5) and (4,4) and the worst performing sets are (3,2) and (5,4). However, as it is shown in this figure the performance of each set is different for different instances, which is because of the structure of the problem.

III. Experiment 3: subtours_selection_criterion

In the third experiment, we investigate different criterion for $subtours\_selection\_criterion$ while fixing s and r to 3 and 5, respectively. Figure 17 shows the comparison between the farthest distance between subtours and other

criteria for the choice of subtours to be involved in the repair mechanism, where positive values mean GPILS under new settings perform better than the previous ones.

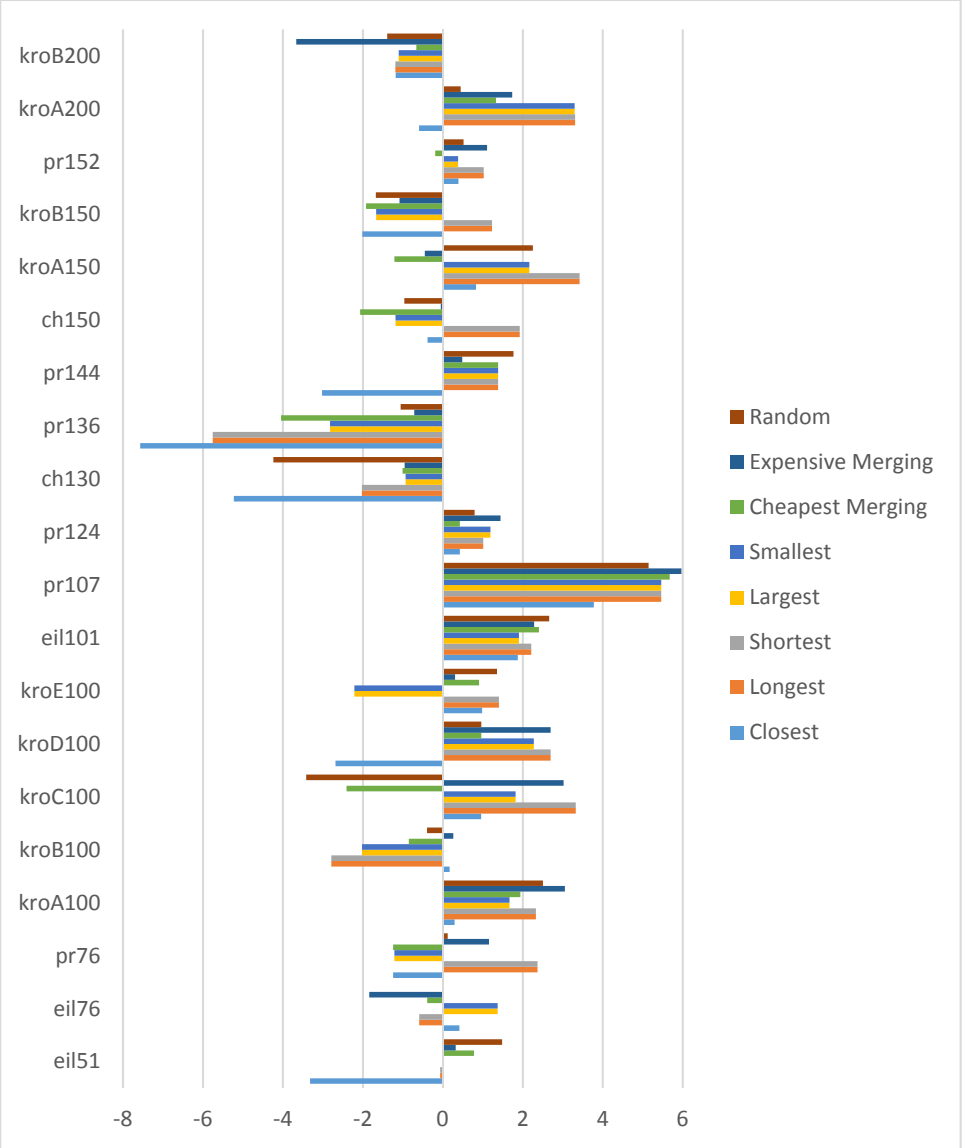


Figure 17 Comparison of subtours_selection_criterion

Overall, on average the best performing sets of values for are longest and shortest, on the other hand, the worst performing sets are closest and cheapest merging criteria however. Moreover, as it is shown in this figure the performance of each choice is different for different instances. Henceforward, the choice of *subtours_selection_criterion* is set to the longest subtours.

IV. Experiment 4: $\left(n_{paths}^{patch}, merging\_criterion\right)$

In this step of the experiment, we shall investigate different n_{paths}^{patch} and $merging_{criterion}$ for the initial paths patching and compare GPILS under new settings the previous one, see Figure 18. Yet again, positive values mean GPILS under new settings perform better than the previous ones. Figure 18 shows the comparison between performance of GPILS given pervious setting and the current one, where the current set is similar to the previous one except for n_{paths}^{patch} and $merging\_criterion$. Note that in the previous setting n_{paths}^{patch} is set to P and $merging\_criterion$ is set to nearest path merging. Thus, we also experimented with n_{paths}^{patch} is set to P and $merging\_criterion$ is set to saving-based path merging, in order to compare their perfomance.

Overall, GPILS given the previous set of parameters given $\left(n_{paths}^{patch}, merging\_criterion\right)$ to $(|P|, saving\_based\_path\_merging)$, $(9, Nearest\_path\_merging)$ and $(6, saving\_based\_path\_merging)$ perform better by %0.31 and %0.24, %0.17, respectively. Yet again, different values for $\boldsymbol{n_{Patch}^{Paths}}$ and $merging\_criterion$ perform different for different instances.

Figure 19 shows the comparison between the performance of GPILS given the current set and either of the two criteria of the $merging\_criterion$; namely, savings-based path merging and nearest path merger, where positive values mean that the nearest path merger performs better than savings-based path merging criteria. This comparison shows that overall, on average, the savings-based path merging procedure performs better than the nearest path merger procedure, and however, comparing the two criteria instance by instance, one can see the difference in performance of these criteria for different problem instances. Although, $(|P|, saving\_based\_path\_merging)$, on average, performs better than other values of n_{paths}^{patch} and $merging\_criterion$. However, we fix n_{paths}^{patch} and $merging\_criterion$ to 6 and saving-based path merging, since we would like to see the performance of other parameters of the patching operation.

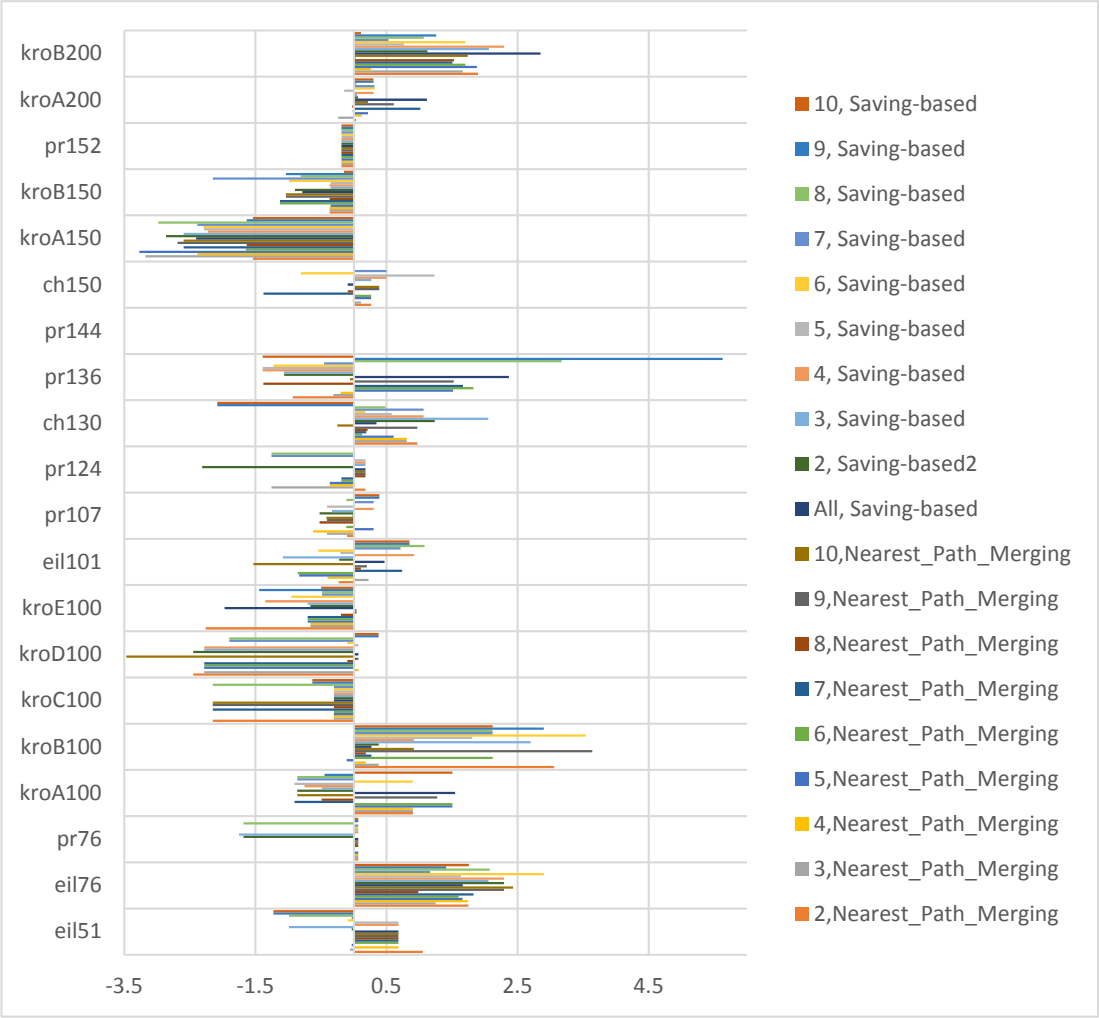


Figure 18 Performance of GPILS given sets of $\left(n_{paths}^{patch}, merging\_criterion\right)$ *and Experiment 3*

V. *Experiment 5:* $paths\_to\_merge\_selection\_criterion$

In the initial patching, the choice of paths to merge could also affect the performance of GPILS. Thus, in this experiment, we investigate different criteria for $paths\_to\_merge\_selection\_criterion$. Figure 20 shows the comparison between GPILS the previous setting and the new one, where in the new set of parameters the only difference is the $paths\_to\_merge\_selection\_criterion$. Overall, when $paths\_to\_merge\_selection\_criterion$ is set to cheapest paths merging cost, GPILS performs better. Thus, in the next experiments, we shall fix this criterion to the cheapest paths merging cost.

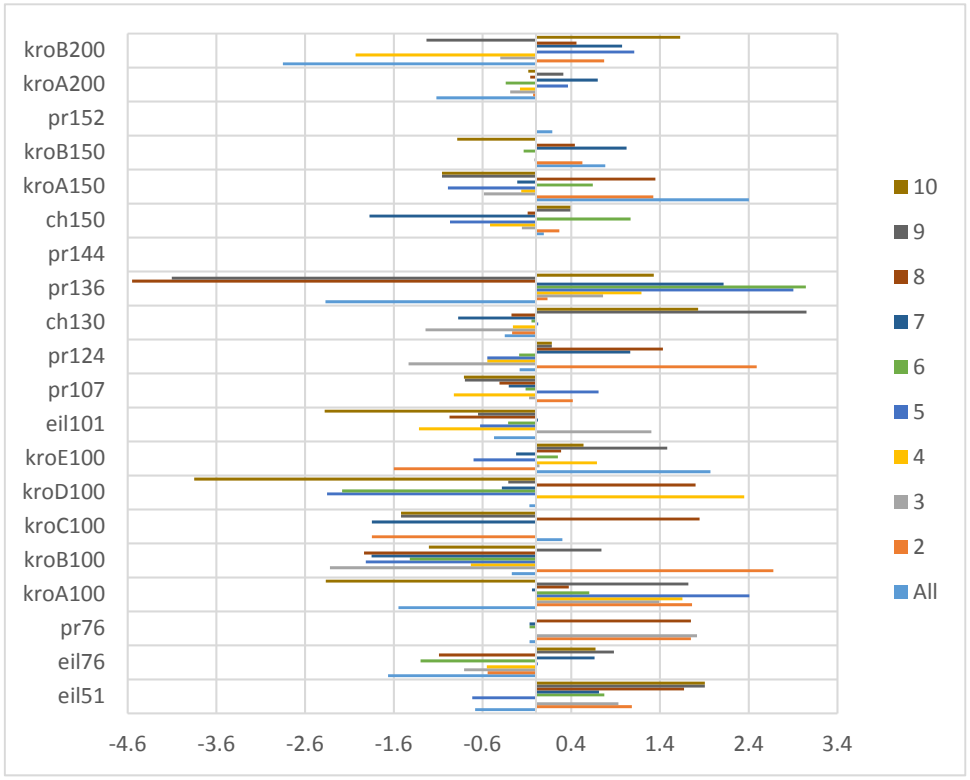


Figure 19 Performance of GPILS given $\left(n_{paths}^{patch}, saving\_based\_path\ \_merging\right)$ *vs. GPILS given* $\left(n_{paths}^{patch}, nearest\_Path\_merging\right)$

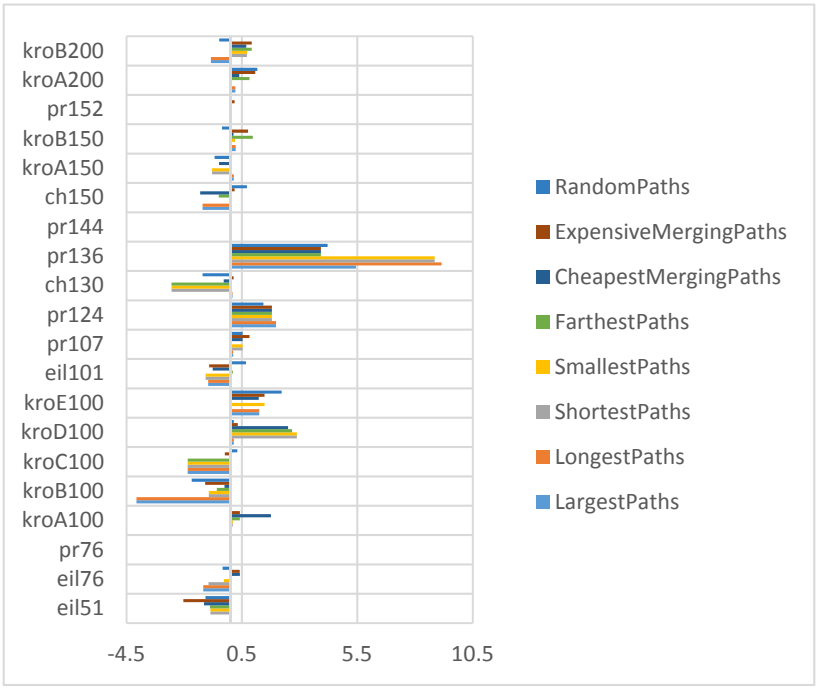


Figure 20 Performance of GPILS given $paths\_to\_merge\_selection\_criterion$

VI. Experiment 6: $paths\_to\_patch\_selection\_criterion$

In this step, we experiment the choice of paths involved in the iterative path patching; namely $paths\_to\_patch\_selection\_criterion$. In general, smallest paths to patch performs better than the other criteria.

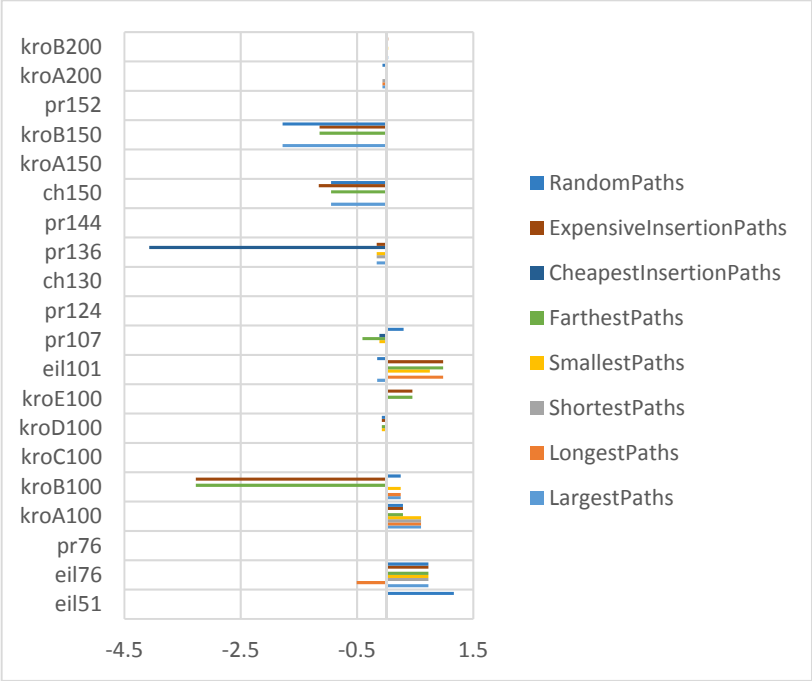


Figure 21 Performance of GPILS given $paths\_to\_patch\_selection\_criterion$

As it can be seen in Figure 21, GPILS given different criterion for $paths\_to\_patch\_selection\_criterion$ performs differently on different problem instances, which could be because of their structure.

VII. Experiment 7: PIH vs AP

As it was mentioned earlier, an initial seed, i.e. infeasible solution, with large number of subtours lead to higher diversity and lower convergence of the search, e.g. AP-relaxation of TSP. On the other hand, initializing the seed with smaller number of subtours could lead to lower diversity and quicker convergence. In this section, we experiment with initialising the initial seed using either AP or *PIH*. Note that we used $(|P|, saving\_based)$ for $\left(n_{paths}^{patch}, merging\_criterion\right)$, since they were best performing values given the parameters set under consideration. Figure 22 shows the computational time(s) of GPILS given previous set and the new one (right) and comparison between performance of GPILS given before mentioned set versus the

new set (left), where in the new set initialisation of the seed and the bound is set to *PIH* and number of initial subtours N_{sbt} is set to a range between 3 and 20.

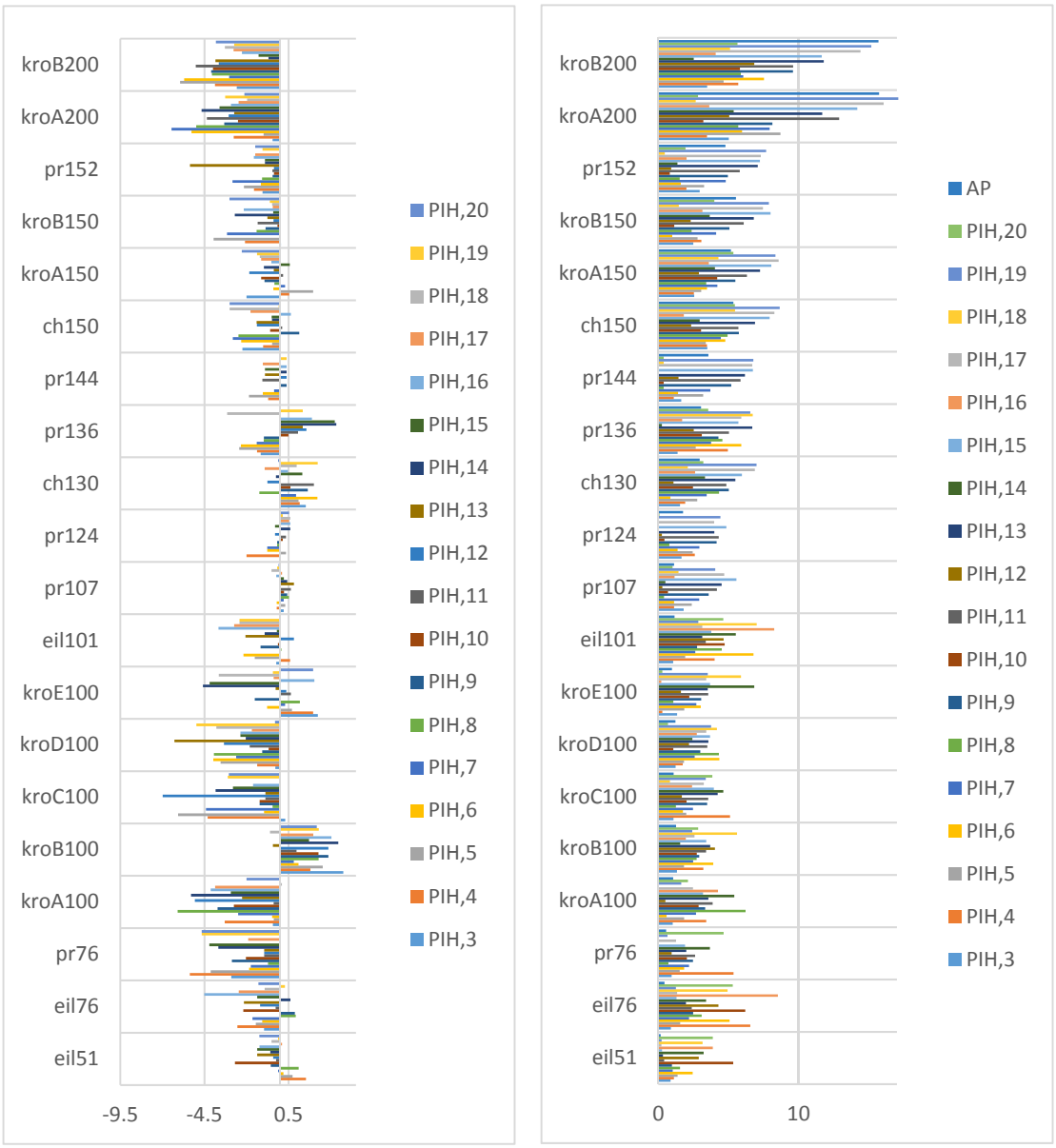


Figure 22 Performance and computational time (s) of GPILS given IM set to AP vs IM set to (PIH,N_{sbt})

Note that each bar in the figure on the left presents performance of GPILS given previous set versus the new set where *IM* is set to *PIH* and $N_{sbt} = \{3, \dots, 20\}$ and each bar on the right shows the computational time of each experiment. In the figure on the left, positive values mean that initialising the seed given *PIH* and N_{sbt} performs better than initialising the seed with AP-relaxation. Moreover, other parameters of the *PIH* namely *DRC*, *Imp* and *NS* are set to savings heuristic, off and 3-opt, respectively.

In terms of quality of the solution, as it can be seen in the figure overall, with these settings, GPILS given the previous set where *IM* is set to AP performs better than the new set. In terms of computational time, in general, for small problem instances GPILS given *IM* set to AP performs faster than GPILS given *IM* set to *PIH*, on the other hand, for larger problems GPILS given *IM* set to *PIH* performs faster than AP. As for N_{sbt}, we cannot make a general conclusion since for different instances the performance is different.

However, since we would like to experiment with other parameters of *PIH*, in the next experiments *IM* is set to *PIH*. As for N_{sbt}, we set it to three, since on average, given the current set, it performs better than other values of N_{sbt}.

VIII. Experiment 8: PIH-DRC

In order to investigate the performance of *DRC*, i.e. decision rule to construct the initial subtours, in the current set we only change the *DRC* and compare the performance of GPILS with the current set without any local improvement, see Figure 23, to isolate the effect of changing *DRC* without any interference of other factors. As it can be seen in the figure, different *DRC* rules perform differently for different problems. However, overall the farthest insertion heuristic performs better than the others. Thus, in the next experiment, we set *DRC* to farthest insertion heuristic.

IX. Experiment 9: PIH - NS

In this section, we investigate the performance of GPILS under the previous setting with different improvement mechanism to locally improving the initial seed obtained by *PIH*. Thus, in the new setting, Imp is set to on, *IMP* is set to local search and *NS* is set to either 2-opt or 3-opt. Later, their performance is compared to the previous setting where the initial seed obtained by *PIH* has not been locally improved, see Figure 24. Note that positive values mean that GPILS performs better when the initial seed, obtained by *PIH*, is improved locally. Overall, on average given the current set, when Imp is set to off, GPILS performs better.

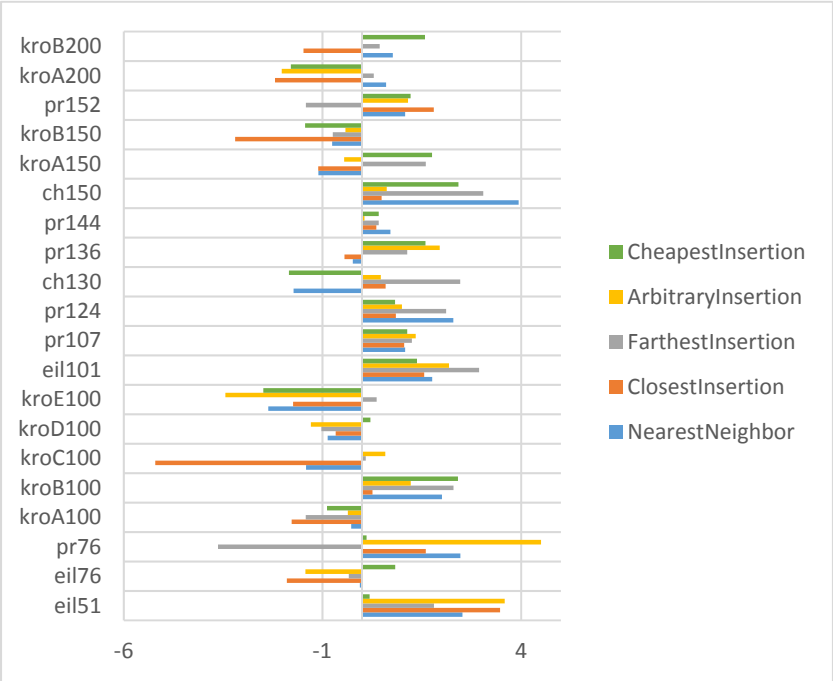


Figure 23 Performance of GPILS given IM set to PIH and (DRC) vs. Experiment 7

X. Experiment 10: T2M and reinforced improvement

With respect to locally improving the intermediate infeasible solution, we shall experiment with T2M and reinforced improvement. To do so, a comparison is made between GPILS given settings, in experiment 7 and 9, and previous setting with different local improvement mechanism for the different intermediate infeasible solutions, see Figure 25 and Figure 26. Figure 26 shows this comparison when the seed is initialised by AP given the previous setting in experiment 7 and Figure 25 shows this comparison when the seed is initialised by *PIH* given the previous setting in experiment 9.

As it can be seen from Figure 25 and Figure 26, when the seed is initialised by *PIH*, GPILS with the combination of T2M set to 2-opt and reinforced improvement set to 3-opt, on average, performs better than the other settings, on the other hand, when the seed is initialised by AP, overall GPILS with T2M set to 3-opt performs better. Figure 27 shows the comparison between GPILS in experiments 7 and 9 with T2M set to 3-opt. As it can be seen GPILS performs differently when the seed is initialised either by AP or *PIH*. However, in the next experiment we shall fix *IM* to AP and T2M to 3-opt since GPILS given this set performs better.

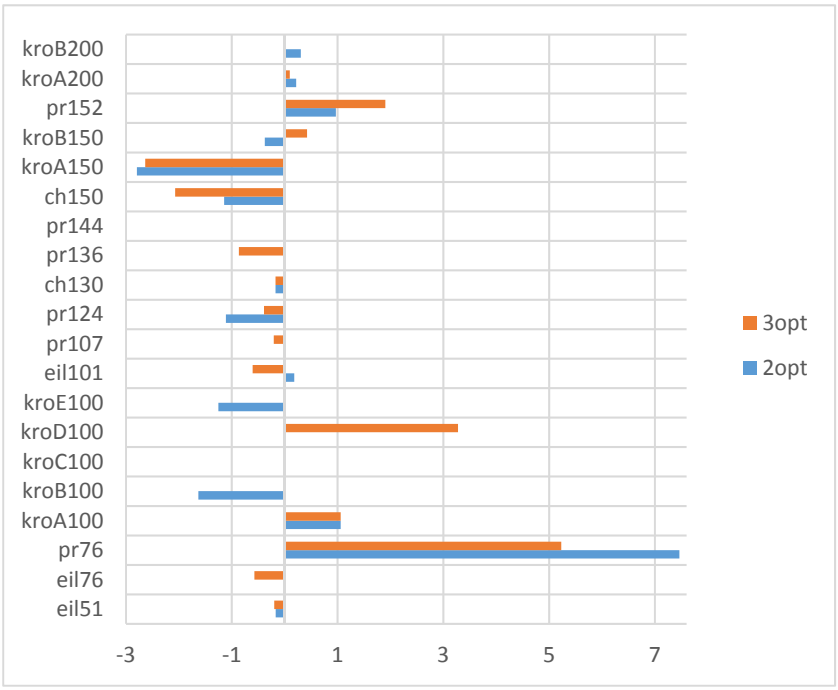


Figure 24 Performance of GPILS given IM set to PIH and (NS) vs. Experiment 8

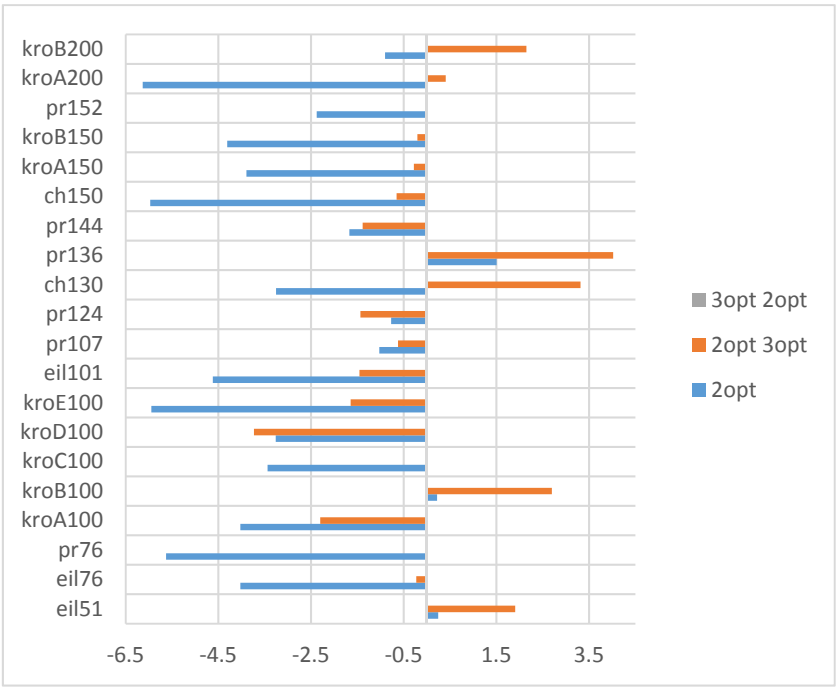


Figure 25 Performance of GPILS given IM set to PIH and (T2M, reinforced improvement) vs. IM set to PIH and T2M set to 3-opt

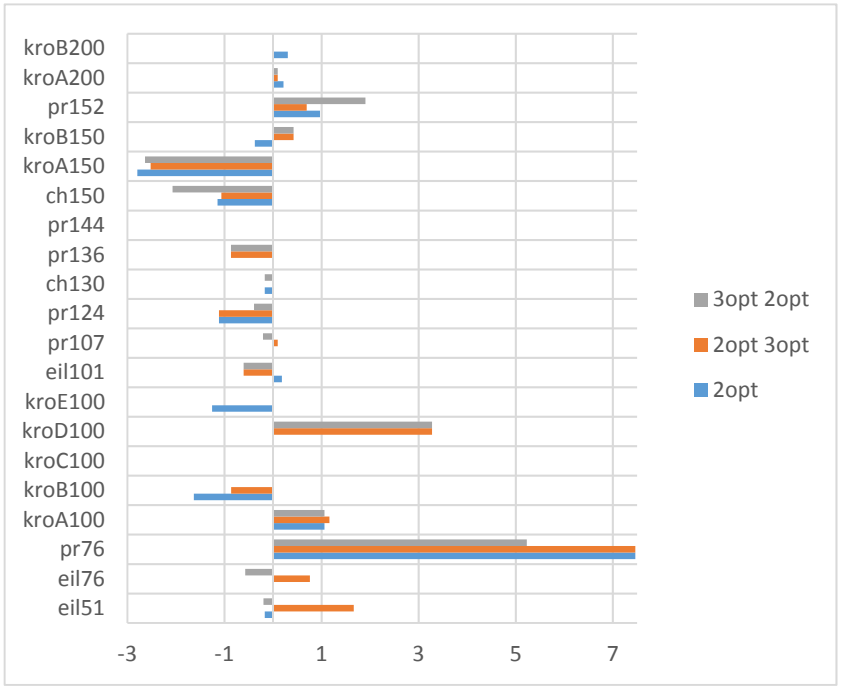


Figure 26 Performance of GPILS given IM set to PIH and (T2M, reinforced improvement) vs. IM set to AP and T2M set to 3-opt

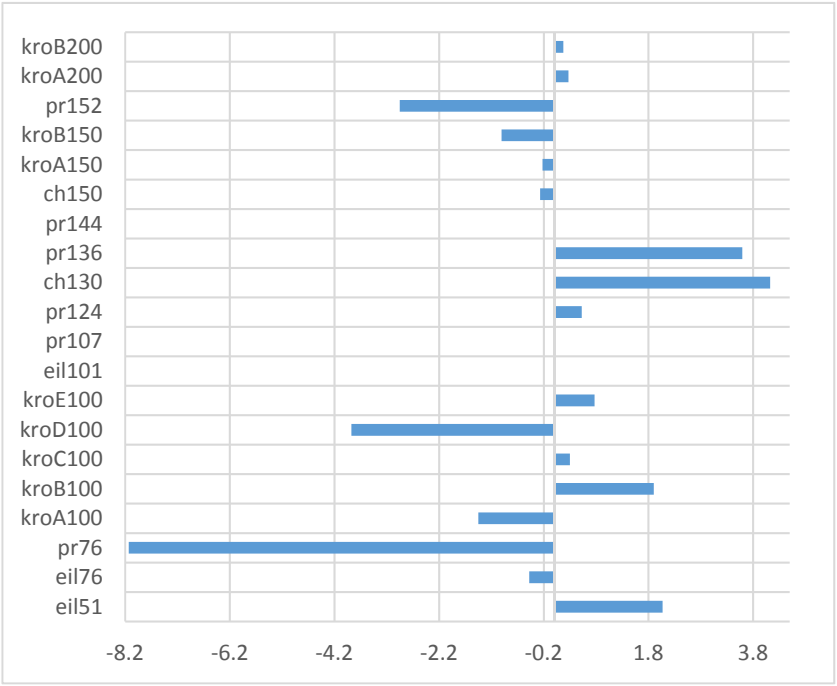


Figure 27 Performance of GPILS given T2M set to 3-opt and IM set to AP vs. IM set to PIH

XI. Experiment 11: PM

In the previous experiments, the bounding scheme, namely the primal bound, was not considered. However, in this experiment we shall consider the bounding scheme,

meaning that we exclude neighbour outside the bounds, see Figure 28. Note that the PM method is set to different construction heuristic and improved by 3-opt local search. Surprisingly, GPILS given the current set without the bounding scheme preforms better, note however that it could be different given different settings. Moreover, given current settings with consideration of different settings of PM, performance of GPILS is quite similar, thus we only present the comparison between performance of given the previous setting and the current settings with consideration of different settings of PM in Figure 28.

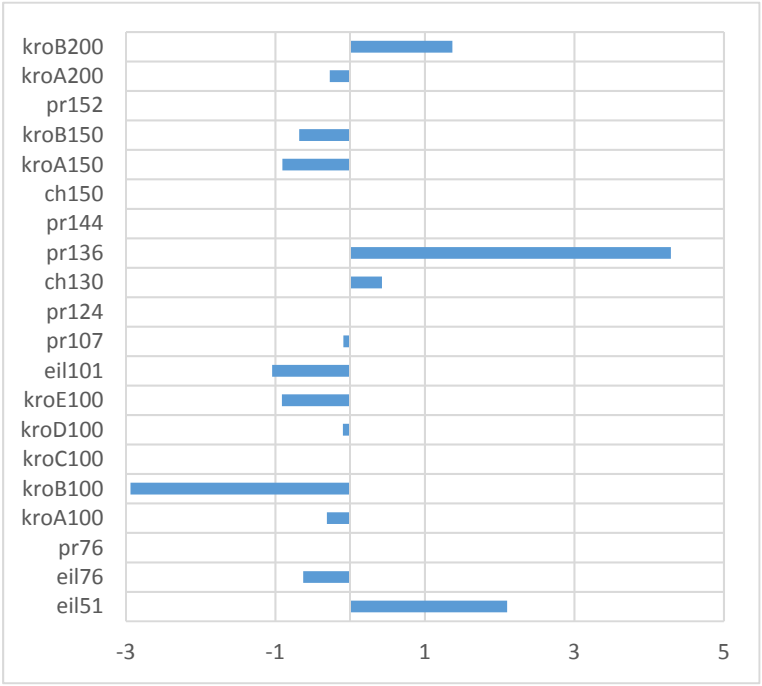


Figure 28 Performance of GPILS with consideration of the primal bound vs. without consideration of the primal bound

We also compare the performance of GPILS given the current set with consideration of the primal bound versus the performance of each of the primal bound, see Figure 29. As it can be seen, GPILS can generate better quality solutions than the primal bound.

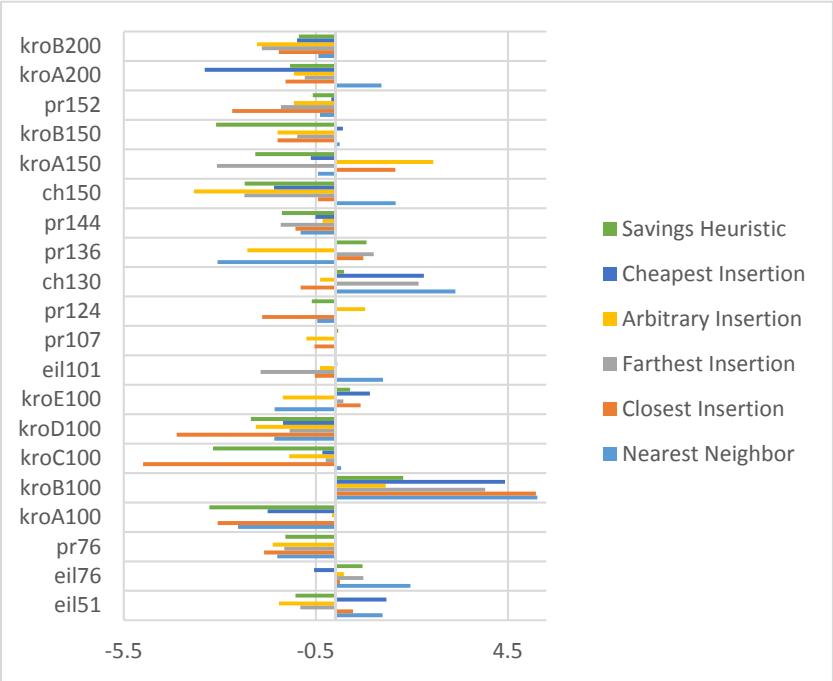


Figure 29 Performance of GPILS with consideration of the primal bound vs. the primal bound PM

3.8. Conclusion

In this chapter we investigated possibility of searching in the infeasible solutions space by a local search-based framework. We refined and enhanced the DLS proposed by Ouenniche et al. (2017) by developing a generic and parameterised infeasible-based local search (GPILS) and pushed it forward to the settings that has not been considered by DLS. The empirical comparison between the proposed GPILS and DLS showed that proposed GPILS can produce better quality solutions much faster. Moreover, a step by step experiment was designed to test several parameters of the GPILS. The experimental design showed that GPILS given different sets of parameters performs differently on different problem instances, which is the case for most primal heuristic solution approaches, thus, there is a need to automate the choice of the parameters of GPILS in order to find the best settings for each problem instance.

The next chapters investigate the case where a hyperheuristic is used to automate and optimise the choice of parameters of GPILS. We also test whether the generated sets of parameters by the proposed hyperheuristic framework can be reused on new and unseen problem instances.

4. A Sequential Hyperheuristic Framework for GPILS

In the previous chapter, we developed a local search-based framework that explores the infeasible search space and progresses toward the feasible space, by reducing the infeasibility, until it reaches a feasible solution. Later, we made a step by step experiment with GPILS given different sets of parameters; where, in the experimental design, each set of parameters was like the previous set except for one or two of the parameters in each set. The analysis showed that since the proposed GPILS given different set of parameters could lead to a different solution to different problem instances, the choice of parameters of GPILS could be automated, instead of being chosen by the analyst.

The aim of this chapter is to automate and optimise the choice of the parameters of said framework, consequently, to find the best possible set of parameters for the given problem instance. Thus, we propose a hyperheuristic to optimise parameters of the GPILS. As it was mentioned in section 2.6, hyperheuristic is high-level mechanism that searches in the space of low-level heuristics or components. In this thesis, the hyperheuristic searches in the space of the parameters of GPILS, see Figure 30, rather than the space of solutions specific TSP instances.

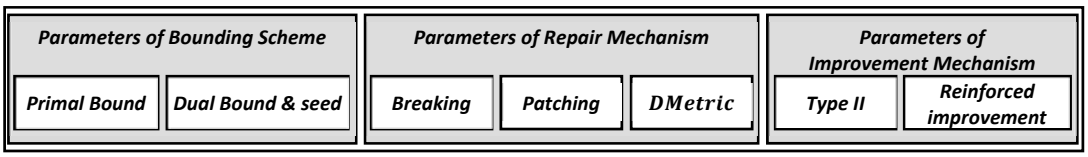


Figure 30 vector of parameters of GPILS

The high-level methodology for searching the parameter space of GPILS could be either a sequential methodology or a parallel one. The contribution of this chapter is developing sequential high-level methodologies to automate and optimise the choice

of parameters of GPILS. To be more specific, we have chosen sequential high-level mechanisms such as SA, TS, and VNS as well as hybrids of these metaheuristics. As for the parallel high-level methodology, we shall develop genetic-based hyperheuristic in the next chapter.

The before mentioned sequential high-level mechanisms start the search with a single set of parameters and search its neighbourhood, in attempt to improve the current set, using a guidance mechanism or search strategy. Implementing these mechanisms require several implementation decisions to be made. Hereafter, we shall discuss the implementation decisions of these metaheuristics for searching the parameter space of GPILS. As the design of these metaheuristics is generic, their implementation for optimising the parameters of GPILS requires a number of decisions to be made. We divide implementation decisions into problem-specific decisions which are common to all metaheuristics and generic decisions which are specific to each metaheuristic.

4.1. Problem-specific decisions for high-level search mechanisms

The decisions common to the implementation of SA, TS, VNS, and our hybrid are as follows: (1) choice of the parameters' space; (2) choice of the form of the objective function; (3) choice of the initial set of parameters of GPILS; and (4) choice of the neighbourhood structure or type of moves to use. These decisions are similar across all these metaheuristics. Hereafter, we shall summarise how these decisions are made for searching the parameter space of GPILS.

Choice of the parameters' space: In principle, all possible vectors of parameters are admissible, an example of a vector of parameters is shown in Figure 30. However, for computational reasons, one might want to reduce the size of the parameter space by imposing bounds on the possible values that some parameters might take or fix the values of some parameters, if considered appropriate. In our experiments, we fixed the values of a number of parameters and for the rest of the parameters we imposed upper bounds on them to keep the computational requirements reasonable.

Choice of the form of the objective function: A variety of functions could be used to discriminate between solutions. In our implementation, we considered the original objective function; namely, the total distance or cost of a solution to the TSP whether infeasible or feasible (primal).

Choice of the initial set of parameters of GPILS: The initial set of parameters of GPILS could either be set by the analyst or be automated. One could automate the choice of an initial set of parameters using a "smart" sequential or parallel random search procedure and select the best vector $param$; that is, the vector leading to the shortest TSP tour. However, for some large problem instances, this option could prove time consuming. One can randomly generate the initial set of parameters, say $param_0$, while respecting the admissible range of parameters' values, if required. This is a sensible choice when a metaheuristic is used for searching the parameter space. Moreover, other could make use of several decision rules such as setting up a parameter to its minimum value, its maximum value, its median, or simply a default value. We propose to automate the choice of $param_0$ by

1. Randomly generating a number of vectors of parameters, evaluating them, and setting $param_0$ to the best vector of parameters,
2. Using a greedy algorithm such as a random descent local search with the number of iterations being the stopping criterion and set to a low number, and setting $param_0$ to the local optimum delivered by random descent, or
3. Running SA for one or several epochs and setting $param_0$ to the best vector of parameters amongst those explored,

where the choice of the best set of parameter is based on the value of the objective function of GPILS given the set of the parameters. In our empirical analysis, we experimented with the first automation process.

Choice of the neighbourhood structure or type of moves to use: A variety of neighbourhood structures could be designed for guiding the search in the parameter space of GPILS. In this thesis, we propose a *generic move* m designed as a function of several parameters along with decision rules which could be formalised as follows:

$$m(nc, DR(EC,\ TC, AC); np); \qquad 30$$

$$DR(EC,\ TC, AC) = (ec,\ tc, ac) \quad 31$$

where np denote the size of the vector of parameters $param$, nc denote the number of entries to change in $param$ amongst the np entries, DR is a *generic parameterised decision rule* with parameters EC, TC and AC, EC is a categorical variable that specifies how to choose the nc entries to modify or change, TC is a categorical variable that specifies the type of change to make to each entry requiring one, and AC is a categorical variable that specifies how to choose the amount by which the value of each of the nc entries to modify will change. The generic parameterised decision rule $DR(EC,\ TC, AC)$ is itself a function whose output is $(ec,\ tc, ac)$, where ec is a vector of np entries that specifies which nc entries to modify or change, tc is a vector of np entries that specifies the type of change to each entry requiring one, and ac is a vector of np entries that specifies the amount by which the value of each of the nc entries to modify will change; to be more specific,

$$ec(i) = \begin{cases} 1 \text{ if entry i is to be changed} \\ 0 \text{ Otherwise} \end{cases} \quad 32$$

$$tc(i) = \begin{cases} 1 \text{ if entry i value is to be increased} \\ -1 \text{ if entry i value is to decreased} \\ 0 \text{ Otherwise} \end{cases} \quad 33$$

and

$$ac(i) = \begin{cases} a_i \text{ if entry i is to be changed} \\ 0 \text{ Otherwise} \end{cases} \quad 34$$

where a_i is an admissible value within the range of parameter i values. In sum, the proposed generic move is a collection of moves. Thus, for any choice of the vector $(EC,\ TC, AC)$, up to np different types of neighbourhood structures could be used to search the parameter space of GPILS. The implementation of the proposed generic move m requires a number of decisions to be made; namely, how to choose the nc entries to modify; how to choose the type of modification; and how to choose the amount of modification. Any of these decisions could be made randomly, using a static decision rule, or using a dynamic decision rule. Hereafter, we shall discuss some of the decision options available.

How to choose the nc entries to modify, or equivalently how to choose EC? The proposed modifications are as follows: one could choose the nc entries to change randomly – we shall refer to this option as the default option (option 0). On the other hand, one could make use of a static rule to choose the nc entries to modify. A range of fifteen static rules are proposed, see Table 28. Finally, one could make use of a dynamic rule to choose the nc entries to modify. Once again, a range of dynamic rules could be designed. For example, one could make use of dynamic rules based on learning, where parameters are modified based on the learning experience accumulated so far and implemented, for example, using the roulette wheel concept, see Appendix E.

How to choose the type of modification to apply, or equivalently how to choose TC? The proposed actions are as follows: one could keep the value of a parameter unaltered (action 1), choose to increase it (action 2), or choose to decrease it (action 3). For each of the nc entries to modify, one could randomly choose amongst these three actions. Note that we refer to this random choice as the default option (option 0).

In our investigation, the choice of nc and the static rules used to modify the nc entries, EC, were defined by a range using options 1, 2, 3 and 4 – see Table 28. In other words, the proposed neighbourhood structure is defined so that only the entries within the permitted range are considered for modification. We proposed two sets of move collections. The first set, called NS1, is a collection of moves that changes only a single range of parameters required for a procedure of GPILS at a time, see Table 29. However, the second set, called NS2, is a collection of moves that changes a range of parameters, see Table 29, meaning that the considered type of modification consists of changing the parameters using either one or two options, see Table 30. After defining which parameters to change, say $\{ec(i)\}$ using either NS1 or NS2, these parameters are changed randomly. In other words, the choice of TC is *option 0* and the choice of AC is also *option 0.*

Another decision rule for changing parameters of the GPILS could involve random choice of the entries to modify and random change in the admissible values to assign. In other words, choosing *option 0* for EC, TC and AC, where the value of nc is predefined by the user.

Option	Description of Type of modification
0	Random modification
1	Only bounding scheme
2	Only breaking procedure of type I move
3	Only patching procedure of type I move
4	Only type II move
5	Only type I move
6	Only choice of INS
7	Only metrics
8	Only infeasible search decisions
9	Modify only primal search decisions
10	Only binary decisions such as PSO
11	Categorical decisions such as PM, IM, SBC, PC, $T2M$, DNS, $Imetric$, PNS, and $PMetric$
12	Only integer variables such as s and r
13	Both binary and categorical decisions
14	Both binary decisions and integer variables
15	Both categorical decisions and integer variables

Table 28 Static rules of modifying EC

```
OptionI= 1;
IF (NeighborhoodStructureI) {
    IF (OptionI < 4)   OptionI ++;
    ELSE OptionI =1;
    DR(OptionI , Option 0, Option 0);
}
```

Table 29 Neighbourhood change strategy considering NS1

```
OptionI = Option;
OptionII  = OptionI;
IF (NeighborhoodStructureII) {
    OptionII  ++;
    IF (OptionII == OptionI OR OptionII > 4) {
        OptionI++;
        OptionII  =  OptionI;
    }
    DR({OptionI, OptionII} , Option 0, Option 0);}
```

Table 30 Neighbourhood change strategy considering NS2

4.2. Generic decisions for high-level search mechanisms

As it was mentioned earlier generic decision are dependent on the structure of the high-level search mechanisms. A summary of each high-level search mechanism and their generic decisions are explained hereafter.

4.2.1. Simulated annealing as a high-level search mechanism

Simulated annealing (SA) is a search procedure based on the annealing process of materials in metallurgy and the underlying thermodynamic laws. Its main search strategy consists of avoiding remaining stuck in a local optimum by temporarily accepting worse solutions with some probability that decreases as the search progresses, for more details refer to section 2.5.1.II. The pseudo-code of the SA algorithm customised to our application is outlined in Table 31.

The implementation of this generic SA algorithm for optimising the parameters of GPILS requires a number of generic decisions to be made which are summarised in the next section.

4.2.2. Generic decisions for SA

Choice of the initial and final temperature: The initial temperature could be chosen by the analyst or using an automated process. In our implementation, we opted for an automated process, where a trial run of the annealing process is performed and the information gathered is exploited in choosing the initial temperature. To be more specific, we computed the initial temperature as follows (Connolly, 1992): $\tau_0 = \delta_{min} + (\delta_{max} - \delta_{min})/2$, where δ_{min} and δ_{max} denote the minimum value and the maximum value of the changes in the objective function over the trial runs, respectively. Note that, the trial run used to compute the initial temperature is the same as the one used to initialise$param_0$. The final temperature τ_f to a small number close to zero, $\tau_f = 0.1$.

Choice of the cooling schedule: The cooling schedule involves several parameters' choices; namely, the number of neighbours to visit at each temperature, say r_t, the

temperature change strategy, the form of the temperature change function $\alpha(t)$ and its parameter(s).

Initialisation Step

Choose an initial set of parameters of GPILS, $param_0$, in the admissible parameter space S and compute the corresponding objective function value $z(param_0)$; that is, the total distance of the TSP tour x_0 constructed by GPILS using the set of parameters $param_0$;

Initialise the best solution found so far, say $(param^*, z^*, x^*)$, by setting $param^* = param_0$, $x^* = x_0$ and $z^* = z(param_0)$;

Choose an initial temperature $\tau^0 > 0$ and set the current temperature $\tau = \tau^0$;

Set the temperature change counter $t = 1$;

Iterative Step

REPEAT until stopping condition = true

Choose the number of neighbours to visit at the current temperature τ, r_τ;

Set the repetition counter $r = 0$;

REPEAT until stopping condition = true // e.g., $r = r_{max}$

Generate randomly a neighbour $param$ of the current seed $param_0$ and call GPILS to evaluate $param$; that is, to compute a primal TSP tour x and its total distance $z(param)$ or $z(x)$;

Compute the change δ in the objective function value: $\delta = z(param_0) - z(param)$;

IF $\delta > 0$ **OR** $Random(0,1) < APF(\delta, \tau)$ **THEN {**

Update the current seed solution $(param_0, z_0, x_0)$; that is, set $param_0 = param$, $x_0 = x$, and $z_0 = z$;

IF $z^* > z(param)$ **THEN {**

Update the best solution found so far $(param^*, z^*, x^*)$; that is, set $param^* = param$, $x^* = x$ and $z^* = z(param)$;}

}

Increment the repetition counter by 1; that is, set $r = r + 1$;

END REPEAT;

Increment the temperature change counter by 1; that is, set $t = t + 1$;

Reduce the temperature τ according to the temperature reduction function α; that is, set $\tau = \alpha(\tau)$;

END REPEAT

Table 31 Pseudo-code of SA as a high-level methodology

In our implementation, we experimented with static value of r_t. As to the temperature change strategy, we experimented with cooling only. With respect to the form of the temperature change function, we proposed a modification of the geometric

temperature reduction function suggested by Kirkpatrick et al. (1983). At the beginning of simulated annealing search, we start with the initial temperature and heat the system by doubling the temperature after each epoch until the epoch counter is less than three or a better solution is found or $\frac{percentage\ accepted\ transitions}{r_t}$ is less than 0.6. Afterwards, cooling the system starts using $\alpha(\tau) = \alpha \times \tau$, cooling schedule proposed by Kirkpatrick et al. (1983). However, if for a number of iterations, say η, if a better neighbour has not been found the temperature is reset to the temperature where the best solution is found and the cooling ratio α is set to β, until an improvement occurs. Whenever an improvement occurs, i.e. a better solution is found the cooling ratio is reset to α. The empirical investigation showed that the appropriate cooling ratio α is to 0.90 and β is 0.8, as for the number of iterations η, it depends on the neighbourhood structure used. In other words, η is set to the size of set of collection of moves used in each neighbourhood structure. For example, if $NS1$ is used to search, the neighbourhood, η is equal to four.

Choice of the transition mechanism: The transition mechanism is specified through the choices of answers to the following questions: (1) how to search the neighbourhood of the current seed solution - randomly or using a suitable method, sequentially or in parallel? and (2) what criteria to use for updating the current seed solution, the first or best improving neighbour? With respect to the first question, we generate randomly and independently r_t neighbours of the same seed solution or several different seed solutions depending on whether the seed solution has been updated or not during epoch t; in sum, each time a neighbour is accepted, the search continues in the neighbourhood of the new seed solution. As to the second question, we experimented with both the first improving neighbour. In addition, we searched the neighbourhood of the current seed solution sequentially when adopting the first improving neighbour strategy for updating the seed.

Choice of the acceptance function (AF): Since SA is not a greedy algorithm, it accepts a neighbour or solution as a new seed because either it is an improving one, or it is a non-improving one but satisfies a second criterion, which could be either deterministic (e.g., Dueck and Scheuer, 1990; Moscato and Fontanari, 1990) or stochastic (e.g., Kirkpatrick et al., 1983; Johnson et al., 1989; Brandimarte et al., 1987). In our

implementation, we experimented with the linear AF proposed by Johnson et al. (1989).

Choice of the stopping criteria: Several stopping criteria can be used such as the 'freezing' state of the system is reached, a prespecified minimum value of the temperature parameter is reached, the number of iterations or temperatures or epochs reaches a prespecified number, computational time exceeds a prespecified time limit, the maximum number of temperature changes without improvement of the current seed is reached, the best objective function value found so far is not updated for a prespecified number of iterations, etc. In our implementation, we experimented with several stopping criteria and opted for both freezing' state of the system is reached and the maximum number of temperature changes without improvement of the current seed is reached. When either of these criteria occurs, the search for the best neighbour stops.

4.2.3. Tabu Search as a high-level search mechanism

Tabu search (TS) algorithms, first proposed by Glover (1986, 1989, 1990), are search procedures that use attribute-based memory structures to constrain and free the search process as needed along with aspiration criteria to override restrictions whenever appropriate. TS main search strategy to avoid remaining stuck in a local optimum is to forbid recent moves for a short while to reduce the likelihood of cycling, using a tabu list or memory, for more details see section 2.5.1.III. The pseudo-code of the short-term memory component-based design of TS algorithms customised to our application is outlined in Table 32. This generic TS algorithm should be customised to our search in the parameter space of GPILS. In the next section, the generic decisions for TS are described.

4.2.4. Generic decisions for TS

Choice of the transition mechanism: In our implementation, the transition mechanism is best described as a constrained steepest descent, where the adjective "constrained" refers to the tabu restrictions.

Choice of the tabu list structure, the way to update it, and its size: Concerning the structure of the tabu list, it is decision variables-oriented in that it is designed to include information on the vector of parameters recently explored. As to the updating of the tabu list, we used a first-in-first-out (FIFO) rule. Finally, concerning the size, we experimented with static size for tabu list**.**

Initialisation Step

Choose an initial set of parameters of GPILS, $param_0$, in the admissible parameter space S and compute the corresponding objective function value $z(param_0)$; that is, the total distance of the TSP tour x_0 constructed by GPILS using the set of parameters $param_0$;

Initialise the best solution found so far, say $(param^*, z^*, x^*)$, by setting $param^* = param_0$, $x^* = x_0$ and $z^* = z(param_0)$;

Specify the aspiration level function and initialise its value;

Choose the tabu list (TL) size and initialise TL to the empty set $\emptyset$;

Set iteration counter I to 0;

Iterative Step

REPEAT until stopping condition = true

Find a move m in the set of applicable moves, say $M(param_0)$, so as to optimise the total distance, say z, of the TSP tour, say x, constructed by GPILS using the set of parameters $param_0$ over the neighbourhood of the current solution, say $N(param_0)$;

IF m or $param = m(param_0)$ is not tabu **THEN**

Update the current seed solution $(param_0, z_0, x_0)$; that is, set $param_0 = param$, $x_0 = x$, and $z_0 = z$;

ELSE

IF m or $param = m(param_0)$ is tabu but the aspiration criterion overrides its tabu status; e.g., $param$ is better than the best vector of parameters found so far **THEN**

Update the current seed solution $(param_0, z_0, x_0)$;

ELSE

Find the best non-tabu move m or neighbour $param = m(param_0)$ – rather than an improving one – in the neighbourhood of the current vector of parameters $param_0$ and update the current seed solution $(param_0, z_0, x_0)$;

Update the tabu list TL;

IF $z(param) < z^*$ **THEN** update the best solution found so far $(param^*, z^*, x^*)$; that is, set $param^* = param$, $x^* = x$ and $z^* = z(param)$;

Increment iteration counter by 1; that is, set $I = I + 1$;

END REPEAT

Table 32 Pseudo-code of TS as a high-level methodology

Choice of the aspiration criteria: Regarding aspiration criteria, we opted for the standard one; namely, the best objective function value achieved for all previous moves.

Choice of the stopping criteria: Several stopping criteria can be used such as the maximum number of iterations is reached, the maximum number of iterations without improvement of the current seed is reached, the computational time exceeds a prespecified time limit, the best objective function value found so far is not updated for a prespecified number of iterations. In our implementation, we experimented with several stopping criteria and opted for the maximum number of iterations without improvement of the current seed is reached.

4.2.5. Variable neighbourhood search as a high-level search mechanism

Variable neighbourhood search (VNS) algorithms, first proposed by Mladenovic and Hansen (1997), are extensions of classical local search algorithms where attempts are made to avoid getting trapped in a local optimum by systematically changing neighbourhood structures during a local search process, for more details see section 2.5.1.IV. The pseudo-code of VNS customised to our application is outlined in Table 33.

This generic VNS algorithm should be customised to our search in the parameter space of GPILS. We divide implementation decisions into problem-specific decisions and generic decisions.

4.2.6. Generic decisions for VNS

What neighbourhood structures to use & how many of them? Any number of neighbourhood structures and a variety of them could be used to guide the search in the parameter space of GPILS. However, using neighbourhood structures with different complexity, starting from simplest to more complex, is preferable. Moreover, one can start the search with neighbourhood structures used to intensify the search and continue with neighbourhood structures used to diversify the search.

Initialisation Step

Choose an initial set of parameters of GPILS, $param_0$, in the admissible parameter space S and compute the corresponding objective function value $z(param_0)$; that is, the total distance of the TSP tour x_0 constructed by GPILS using the set of parameters $param_0$;

Initialise the best solution found so far, say $(param^*, z^*, x^*)$, by setting $param^* = param_0$, $x^* = x_0$ and $z^* = z(param_0)$;

Choose a set of neighbourhood structures to use and specify the order according to which they will be used, say $\{N_s; s = 1, \dots, s_{max}\}$;

Choose the local search method to use in exploring neighbourhoods;

Initialise neighbourhood structure counter s to 1;

Iterative Step

REPEAT until stopping condition = true

Randomly generate a neighbour, say $param$, of the current vector of parameters or seed $param_0$ according to the s-th neighbourhood structure;

Explore the s-th neighbourhood of $param$ using the chosen local search method and update $param$ accordingly;

IF this local optimum concerning the s-th neighbourhood $param$ is better than the current seed $param_0$ **THEN**

Update the current seed solution $(param_0, z_0, x_0)$; that is, set $param_0 = param$, $x_0 = x$, and $z_0 = z$;

IF $z(param) < z^*$ **THEN** Update the best solution found so far $(param^*, z^*, x^*)$; that is, set $param^* = param$, $x^* = x$ and $z^* = z(param)$;

Reset neighbourhood structure counter s to 1;

ELSE Increment neighbourhood structure counter s by 1;

END REPEAT

Table 33 Pseudo-code of VNS as a high-level methodology

Choice of the transition mechanism to use: The transition mechanism is specified through the choices of answers to the following questions:

(1) How to search a specific neighbourhood of the current seed solution?

In our implementation, we used random descent local search to search a proportion of the neighbourhood of the current seed.

(2) What criteria to use for updating the current seed solution?

Concerning criteria to update the current seed, we experimented with best-improving neighbour amongst equal or improving neighbours.

(3) What criteria to use for changing neighbourhoods?

Concerning changing the neighbourhood structure, we experimented with $Sequential\_Neighborhood\_Change$ strategy.

(4) In which order to search the neighbourhoods?

The neighbourhood structures are ordered based on their complexity, intensification and diversification level, in non-decreasing order.

Choice of the Stopping Criteria: Our choice is similar to the one made above for TS.

4.3. Hybrid hyperheuristics

Each of the before mentioned sequential high-level mechanisms makes use of different intensification and diversification strategies, hence, one can hybridise them to create a better balance between these strategies. As it was mentioned in the previous section, the implementation of the proposed HH-GPILS framework for the TSP requires two types of decisions, namely problem-specific and generic. One might propose a hybrid high-level framework by combining their generic decisions. We proposed several hybrid hyperheuristics; namely hybrid of SA and TS, see Figure 31; hybrid of VNS and TS, see Figure 32; and a hybrid of SA, VNS, and TS, see Figure 33. For the first two hybrids, we incorporated a tabu list (TL) into SA and VNS, with the same specification of the one used in TS. Furthermore, we used aspiration criteria used in TS. In other words, if the new neighbour $param$ is Tabu but it satisfies the aspiration criteria, its tabu status will be overwritten. As for the third hybrid, in addition to the TL and the AC, we incorporated the neighbourhood change strategy of the VNS.

4.4. Hyperheuristics with intensification strategy

As it was mentioned in section 0, intensification strategies are used to search the promising areas around local optima. One approach is to restart the search from a (perturbed) local optima and improve it using different neighbourhood structures, for several iterations, looking for a better neighbour.

Initialisation Step

Choose an initial set of parameters of GPILS, $param_0$, in the admissible parameter space S and compute the corresponding objective function value $z(param_0)$; that is, the total distance of the TSP tour x_0 constructed by GPILS using the set of parameters $param_0$;

Initialise the best solution found so far, say $(param^*, z^*, x^*)$, by setting $param^* = param_0$, $x^* = x_0$ and $z^* = z(param_0)$;

Choose an initial temperature $\tau^0 > 0$ and set the current temperature $\tau = \tau^0$;

Adjust the temperature change counter $t = 1$;

Specify the aspiration level function and initialise its value;

Choose the tabu list (TL) size and initialise TL to the empty set $\emptyset$;

Choose a set of neighbourhood structures to use and specify the order according to which they will be employed, say $\{N_s; s = 1, \dots, s_{max}\}$;

Initialise neighbourhood structure counter s to 1;

Initialise the restart counter $\Re$ to 0;

Iterative Step

REPEAT until stopping condition = true

Randomly generate a neighbour, say $param$, of the current vector of parameters or seed $param_0$ according to the s-th neighbourhood structure, at the current temperature τ, r_τ;

Set the repetition counter $r = 0$;

REPEAT until stopping condition = true // e.g., $r = r_{max}$

Generate randomly a neighbour $param$ of the current seed $param_0$ and call GPILS to evaluate $param$; that is, to compute a primal TSP tour x and its total distance $z(param)$ or $z(x)$;

$param = m(param_0)$ is tabu THEN

IF $param = m(param_0)$ is tabu but the aspiration criterion overrides its tabu status **THEN**

Update the current seed solution $(param_0, z_0, x_0)$;

ELSE Find the best non-tabu move m or neighbour $param = m(param_0)$ – rather than an improving one – in the neighbourhood of the current vector of parameters $param_0$ and update the current seed solution $(param_0, z_0, x_0)$;

Update the tabu list TL;

Compute the change δ in the objective function value: $\delta = z(param_0) - z(param)$;

IF $\delta > 0$ **OR** $Random(0,1) < APF(d, \tau)$ **THEN{**

Update the current seed solution $(param_0, z_0, x_0)$; that is, set $param_0 = param$, $x_0 = x$, and $z_0 = z$;

IF $z^* > z(param)$ **THEN{**

IF $z(param) < z^*$ **THEN{**

Update the best solution found so far $(param^*, z^*, x^*)$; that is, set $param^* = param$, $x^* = x$ and $z^* = z(param)$;

Set $\Re_{Best} = \Re$,

Set $\tau^* = \tau$ and $\alpha = 0.9$;**}**

Reset neighbourhood structure counter s to 1; **}}**

Increment the repetition counter by 1; that is, set $r = r + 1$;

END REPEAT;

Increment the temperature change counter by 1; that is, set $t = t + 1$;

Increment neighbourhood structure counter s by 1;

IF $s_{max} < s$ **THEN{**

Reset neighbourhood structure counter s to 1;

IF $\Re - \Re_{Best} < \Re_{max}$ **THEN{**

Restart the search with the best solution found so far; that is set $param = param^*$, $x_0 = x^*$, and $z_0 = z^*$;

Reset the temperature τ to τ^*;

Increment restart counter $\Re$ by 1;

Set $\alpha = \beta$;**}**

ELSE Return the best solution found so far $(param^*, z^*, x^*)$;

ELSE Reduce the temperature τ according to the temperature reduction function: $\tau = \alpha\tau$;

END REPEAT

Table 34 Pseudocode for hybrid of SA, TS, and VNS with restart

Thus, to exploit the promising areas in the neighbourhood of the current local optima, we used an intensification strategy for all the proposed sequential HH-GPILS where it restarts the search with the best solution found so far, after several iterations with no improvements, see Table 34. Note however that the neighbourhood structure used allows the search to explore new paths in the search space. Therefore, the intensification is likely to reach new local optimum. Moreover, we added new stopping criterion to the previous stopping criteria chosen for each of the sequential based HH-GPILS, namely stopping the search when no better solution is found for a number of restarts, say $\Re_{max}$.

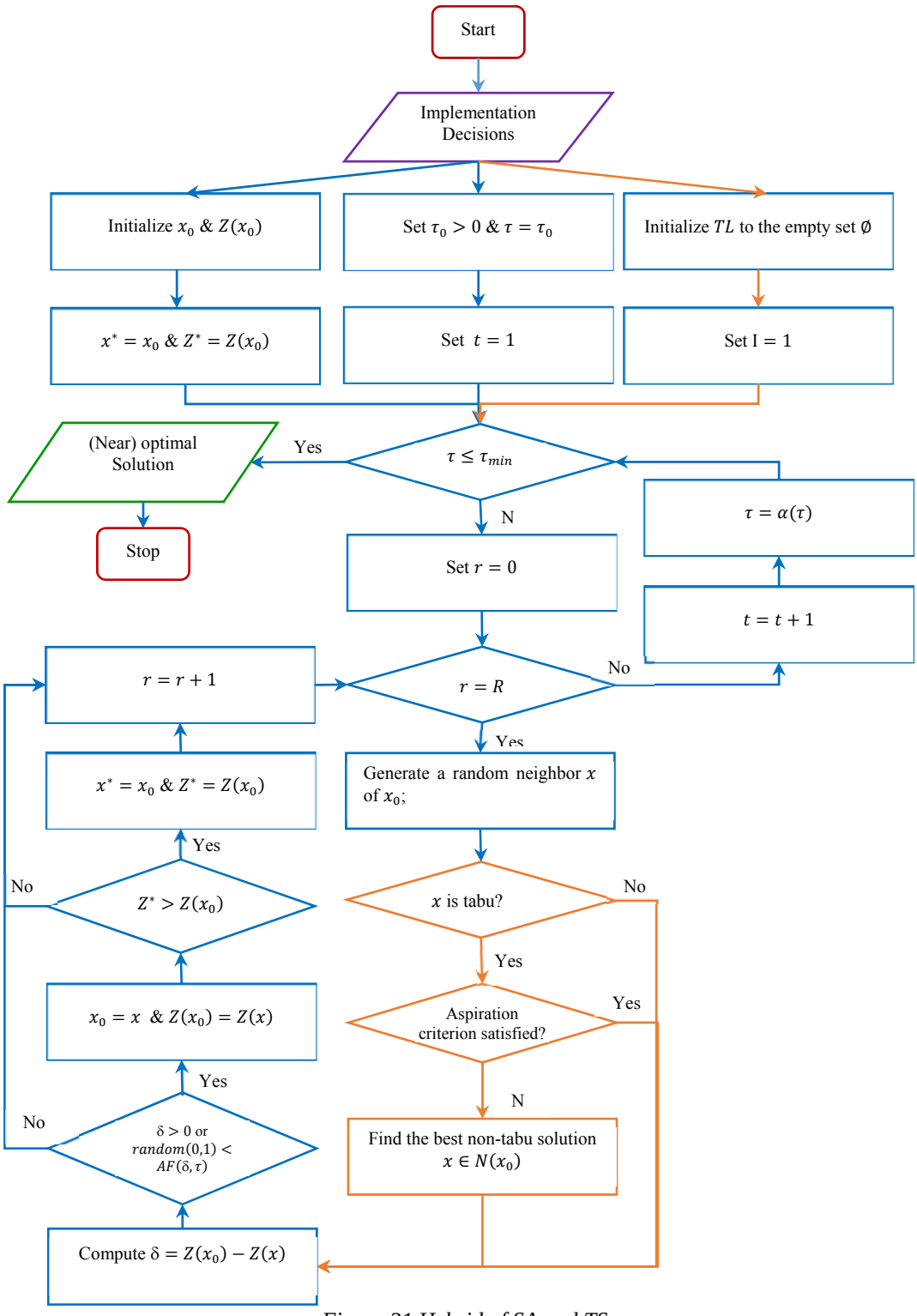


Figure 31 Hybrid of SA and TS

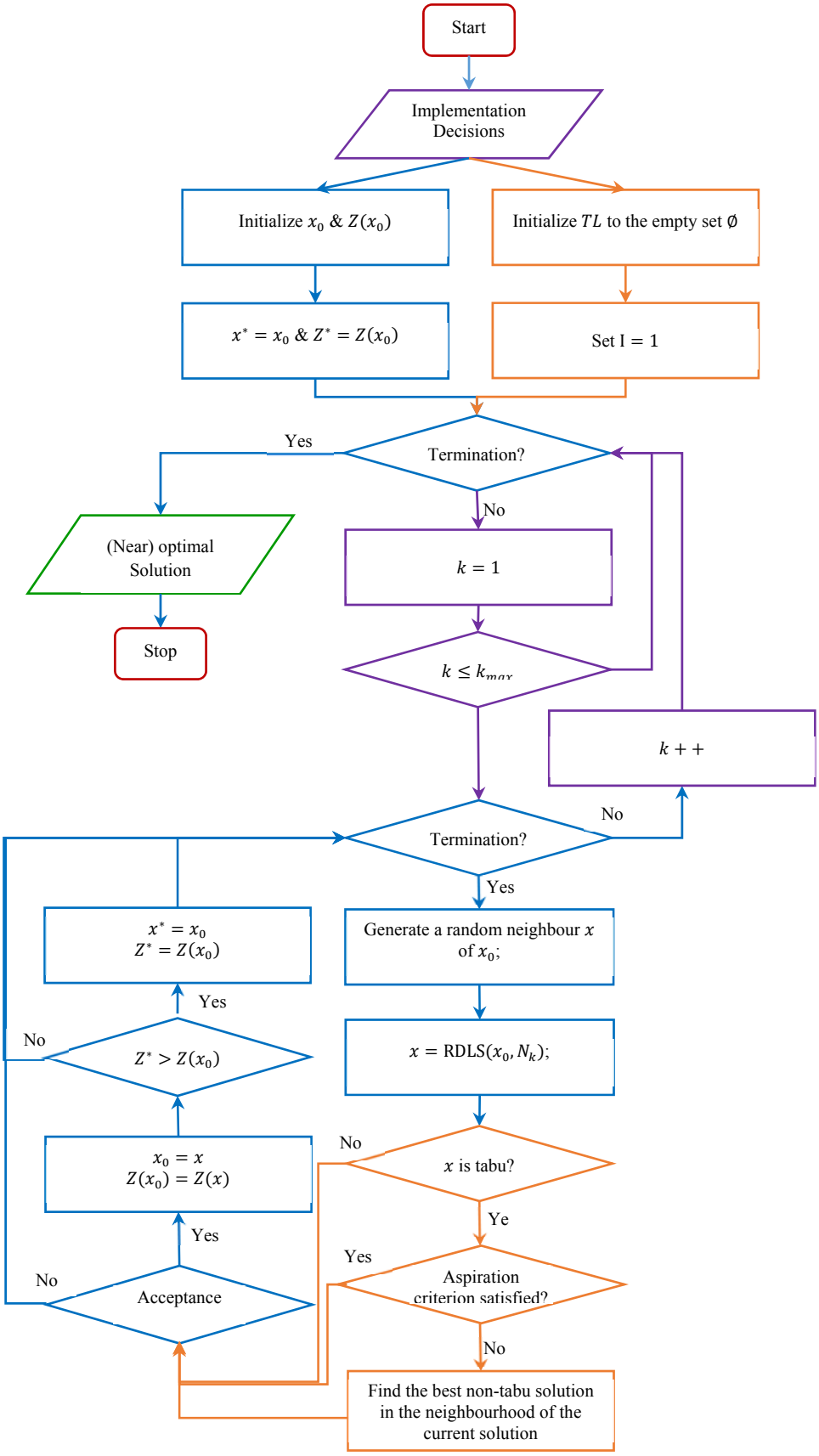


Figure 32 Hybrid of VNS and TS

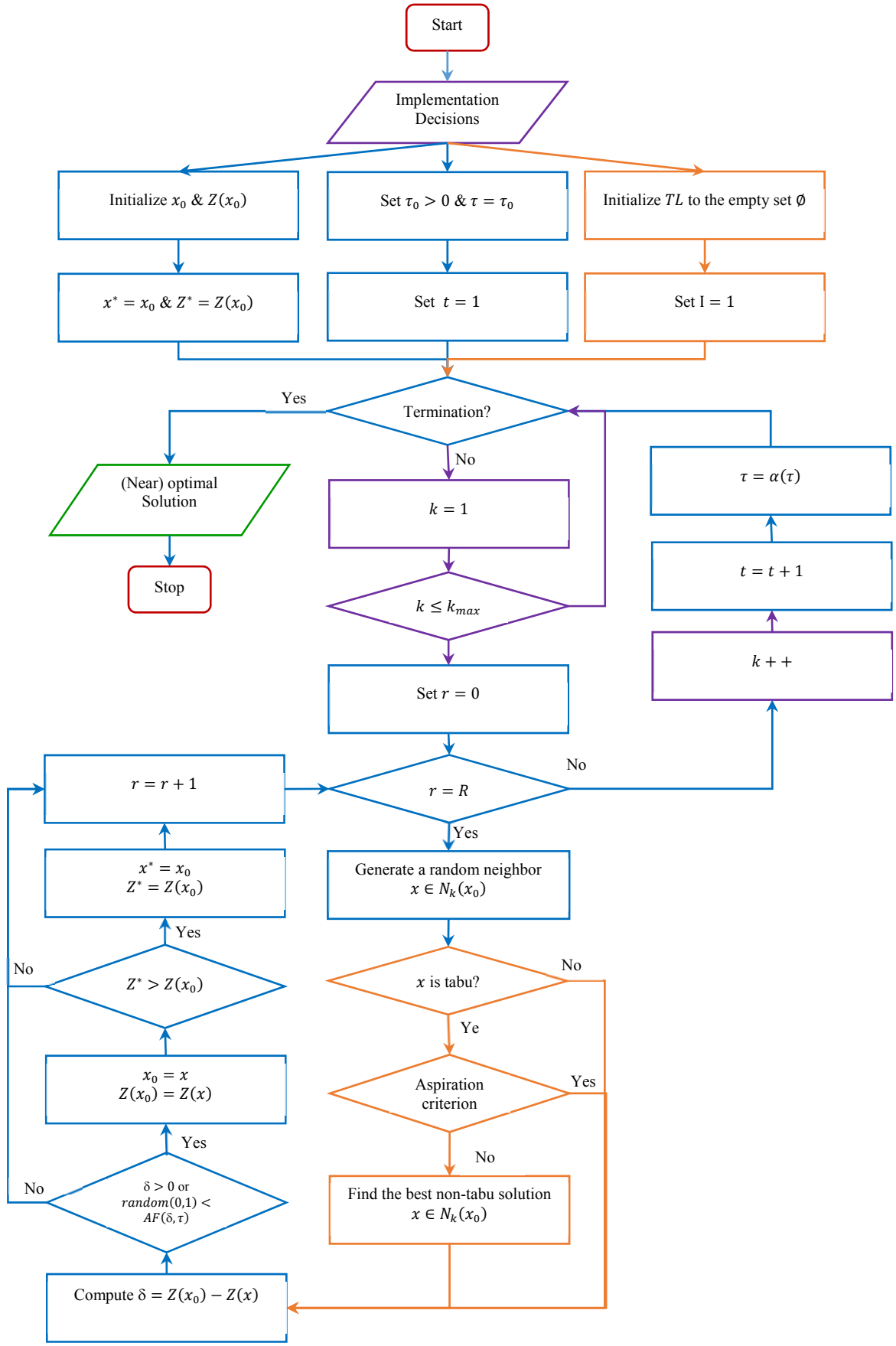


Figure 33 Hybrid of SA, TS, and VNS

4.5. Empirical investigation

This section presents a comparative analysis of the proposed sequential hyperheuristics, which automate the choice of parameters of GPILS. We investigate the performance of the proposed methods by following three stages. First, we show the empirical results of each method separately and later we compare their performance. Second, we compare the proposed HH-GPILS with the DLS developed by Ouenniche et al. (2017), which has some similarities (i.e. the infeasible nature of the search) with our proposed method. This comparison proves that using a hyperheuristic to automate the set of parameters could lead to better performing heuristics in comparison to the tailor-made ones. Finally, we show a comparison between the best three sequential HH-GPILS and the relevant benchmark. This comparison showed that HH-GPILS is a promising design.

4.5.1. Experimental setup

In this chapter, we experimented with sequential methodologies, namely SA, TA, VNS and their hybrids. We divided the implementation decisions of these hyperheuristics to problem-specific and generic decisions, as it was explained throughout this chapter. These decisions are shown in Table 35. The choice of each of these decisions is specified hereafter.

For the *choice of the parameters space of GPILS* in this empirical investigation, we reduced the size of the admissible vector of parameters to keep the computational requirements reasonable. Note that this limitation of search space might reduce the quality of the solution obtained. In our empirical investigation, we imposed bounds on some of the parameters, while fixing the values of other parameters. Parameters of HH-GPILS are as follows:

Parameters of the bounding scheme

- PM: Nearest neighbour; arbitrary insertion; nearest insertion; farthest insertion; cheapest insertion; Clarke and Wright; nearest merger
- *IM*: {AP, *PIH* }
- Parameters of *PIH*

Type of Decisions	Decision	High-Level Methodology
Problem-specific	Parameters' space	SA/ VNS / TS
	Initial set of parameters of GPILS	SA/ VNS / TS
	Neighbourhood structure or type of moves to use	SA/ VNS / TS
	Optimisation function	SA/ VNS / TS
Generic	Initial temperature	SA
	Cooling schedule	SA
	Transition mechanism	SA/ VNS / TS
	Acceptance function (AF)	SA
	Tabu list structure	TS
	Aspiration criteria (AC)	TS
	Neighbourhood structures to use and how many of them	VNS
	Stopping criteria	SA/ VNS / TS

Table 35 HH-GPILS high-level decisions

- N_{Sbt}: $1, \dots, {}^{n}/_{3}$
- $CO = 1$
- DRA: $\mathcal{K}$-means ($\mathcal{K} = N_{Sbt}$)
- DRC: Construction heuristic similar to PM
- Imp:{0, 1}
- IM: Classic local search
- NS: 2-opt; 3-opt; Or-opt

Parameters of Type I move

- Breaking operation
  - s: $1, \dots, 5$
  - $subtours\_selection\_criterion$: Random; shortest/ largest subtours; smallest/ largest subtours; closest/ farthest subtours; cheapest/ most expensive cost of merging pair of subtours
  - r: $1, \dots, 5$ (if $r > |S|$ then $r = |S| - 1$)
  - $arcs\_to\_break\_selection\_criterion$: K-NN ($K = 5$; if $k \leq r$ then $K = r + 2$)
- Patching operation

- *type_of_patching_operation*=1
- Initialisation step
    - n_{paths}^{patch}: $1, \dots, P$
    - $paths\_to\_merge\_selection\_criterion$: Largest / smallest; longest/ shortest; closest/ farthest; cheapest/expensive merging cost
    - $merging\_criterion$: Saving procedure; nearest merger
    - $type\_of\_implementation = 0$
- Iterative patching
    - n_{paths}^{insert} :[0,1]
    - $paths\_to\_patch\_selection\_criterion$: Largest / smallest; longest/ shortest; closest/ farthest;
    - $patching\_criterion$: Cheapest Insertion
    - $type\_of\_implementation = 0$
- $patching\_operation\_performance\_criterion$: Cost of the subtour

Parameters of Type II move

- *T2M*: 2-opt; 3-opt; ; Or-opt
- $component\_improvement\_mechanism$: Local search
- $component\_performance\_metric$: Cost or total distance of the component

Note that, we also used 'reinforced improvement' with some probability (0.7). In other words, after improving the infeasible component by the chosen *T2M*, we improved the infeasible component using a different *T2M,* again.

Other parameters of GPILS

- $IMetric:$ Cost or total distance of the component
- INS: IBN

Parameters of the primal space exploration

- $explore\_primal\_space = 0$

For the choice of the neighbourhood structures, we experimented with NS1 and NS2 to search the parameter space of GPILS. As for the decision rules, we experimented with several decision rules, see Table 36.

DR	Option
EC	$NeighborhoodStructureI$ or/and $NeighborhoodStructureII$
TC	option 0
AC	option 0

Table 36 Decision rules

The rest of the decisions presented in Table 35 are explained in the previous sections, however, their parameters are as follows.

- Number of trial runs: 20
- $r_{max} = 40$
- $\alpha = 0.9, \beta = 0.8$
- $\tau_f = 0.1$
- $|TL| = 15$
- $\Re_{max} = 3$

In order to understand the effect of initialising the infeasible solution using either *AP* or *PIH*, first, for all proposed HH-GPILS methods, we experimented with fixing IM to either *AP* or *PIH*. The empirical results show that initialising the infeasible solution with *PIH* has higher quality than initialising with *AP*. Furthermore, for the proposed SA and TS based HH-GPILS, we experimented with NS1 and NS2 separately and for the VNS-based HH-GPILS we experimented with both NS1 and NS2, one after the other. In general, $NS2$ perform better than $NS1$, since has higher level of diversity.

4.5.2. Experimental results

The statistics presented in this section are the performance of the hyperheuristic calculated as average percentage increase over the optimal solution (i.e. $\frac{tour\ cost - known\ optimum}{known\ optimum} \times 100\%$) and also the computational time (in seconds) for a number of runs (i.e. in this investigation 5 runs). Note that is measured as the performance of the proposed HH is measured as average percentage increase over the optimal solution. In Table 37-40 and 43-44, the first column shows the instances solved, the rest of the columns are divided into three sets showing the results for three experiments. For the first experiment, we fixed *IM* to AP, for the second experiment we fixed *IM* to *PIH*, and for the third we did not fix IM. In each set of experiments,

the first column shows the results where only NS1 neighbourhood structure is used in HH-GPILS and the second column shows the results where only NS2 neighbourhood structure is used. In Tables 41-42 and 45 to 52, each of the columns show the results of the following experiments:

1. First experiment: parameter *IM* is fixed to AP;
2. Second experiment: parameter *IM* is fixed to AP;
3. Third experiment: parameter *IM* is not fixed.

In these three experiments, NS1 and NS2 neighbourhood structures are used, one after another. Hereafter, we shall discuss the performance of the proposed hyperheuristics.

The results obtained by *SA-based HH-GPILS* are shown in Table 37 and 38, reporting its performance and computational time, respectively. From these statistics, one can conclude:

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	0.00%	0.05%	0.09%	0.00%	0.19%	0.00%
eil76	0.22%	0.04%	1.08%	0.45%	1.12%	0.59%
pr76	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.07%	0.00%	0.01%	0.00%
kroB100	0.00%	0.00%	0.07%	0.00%	0.15%	0.00%
kroC100	0.00%	0.00%	0.10%	0.02%	0.18%	0.02%
kroD100	0.00%	0.00%	0.12%	0.06%	0.23%	0.00%
kroE100	0.00%	0.00%	0.04%	0.00%	0.00%	0.00%
eil101	0.79%	0.73%	1.14%	0.70%	1.30%	0.99%
pr107	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
pr124	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%
ch130	0.15%	0.18%	1.06%	0.32%	0.54%	0.36%
pr136	0.04%	0.03%	0.03%	0.00%	0.01%	0.00%
pr144	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ch150	0.35%	0.25%	0.73%	0.52%	0.95%	0.58%
kroA150	0.17%	0.14%	0.58%	0.45%	0.66%	0.40%
kroB150	0.04%	0.02%	0.35%	0.14%	0.24%	0.20%
pr152	0.00%	0.00%	0.07%	0.00%	0.11%	0.04%
kroA200	0.51%	0.68%	0.80%	0.59%	0.74%	0.71%
kroB200	0.59%	0.46%	1.49%	0.92%	1.30%	0.60%
Average	0.14%	0.13%	0.39%	0.21%	0.39%	0.22%
Median	0.00%	0.01%	0.10%	0.01%	0.18%	0.01%
Std	0.23%	0.22%	0.47%	0.28%	0.45%	0.30%

Table 37 Performance of SA-based HH-GPILS

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	184	50	57	110	50	326
eil76	250	143	119	240	57	538
pr76	243	104	76	169	52	434
kroA100	409	293	117	353	148	696
kroB100	407	336	101	282	134	688
kroC100	533	294	252	338	159	593
kroD100	438	390	382	329	203	834
kroE100	372	367	153	382	179	602
eil101	458	453	285	646	177	617
pr107	680	536	229	498	435	452
pr124	512	591	395	568	223	763
ch130	613	1359	438	833	379	1641
pr136	816	1318	326	721	674	825
pr144	733	1104	372	901	360	1551
ch150	878	1360	440	1097	426	2084
kroA150	1255	1167	395	963	503	1777
kroB150	1371	1174	581	785	672	1977
pr152	904	641	729	757	720	1473
kroA200	1230	1330	878	1368	1721	1391
kroB200	1013	1211	537	1055	1848	2702

Table 38 Computation time of SA-based HH-GPILS

- Only considering *PIH* to initialise the infeasible solution (fixing *IM* to *PIH*) one can obtain better quality solutions;
- Furthermore, using NS2 neighbourhood structure to search the infeasible space has a better performance than NS1 neighbourhood structure in terms of quality of the solution, although it is more time consuming than NS1;
- Overall, *SA-based HH-GPILS* using NS2 neighbourhood structure and parameter *IM* fixed to *PIH* is performing better than the others, with average 0.3%, median 0.1% and standard deviation 0.22%;
- As for the computational time, fixing parameter *IM* to *AP* seems to be seems to be more efficient. The reason is that the parameters of *PIH* are not considered in the parameters search space.

Table 39 and 40 report *TS-based HH-GPILS* performance and computational time, respectively. From these experiments following conclusions can be drawn:

- Fixing *IM* to *PIH* has better performance than the others. In other words, initialising the infeasible solution using *PIH* could lead to better results;

- Using NS2 neighbourhood structure provides a better solution in comparison to NS1 neighbourhood structure, however in terms of computational time NS1 is more efficient;
- Overall, *TS-based HH-GPILS* with NS2 neighbourhood structure and parameter *IM* fixed to *PIH* has better performance than the others, with average 0.13%, median 0.0% and standard deviation %0.22;
- Searching the parameters space of the GPDLS using NS1 neighbourhood structure leads to good quality solutions much faster than NS2 and combination of NS1 and NS2;
- Furthermore, *TS-based HH-GPILS* with NS1 neighbourhood structure and parameter IM set to $\{AP, PIH\}$ produces good quality solutions in faster than the other settings.

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	0.23%	0.00%	0.23%	0.19%	0.00%	0.09%
eil76	0.33%	0.45%	1.08%	0.63%	0.37%	0.48%
pr76	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.07%	0.06%	0.00%	0.02%
kroB100	0.00%	0.00%	0.09%	0.05%	0.00%	0.00%
kroC100	0.00%	0.00%	0.09%	0.02%	0.04%	0.02%
kroD100	0.17%	0.00%	0.36%	0.09%	0.08%	0.13%
kroE100	0.00%	0.00%	0.12%	0.00%	0.00%	0.00%
eil101	0.86%	0.83%	1.05%	1.18%	0.92%	1.02%
pr107	0.01%	0.00%	0.05%	0.00%	0.00%	0.00%
pr124	0.02%	0.00%	0.02%	0.00%	0.00%	0.00%
ch130	0.40%	0.20%	1.10%	0.54%	0.54%	0.55%
pr136	0.09%	0.04%	0.25%	0.00%	0.01%	0.04%
pr144	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ch150	0.52%	0.34%	0.65%	0.62%	0.56%	0.64%
kroA150	0.52%	0.16%	0.72%	0.59%	0.48%	0.35%
kroB150	0.02%	0.02%	0.28%	0.31%	0.24%	0.13%
pr152	0.00%	0.00%	0.11%	0.11%	0.07%	0.04%
kroA200	0.55%	0.42%	0.78%	0.76%	0.69%	0.73%
kroB200	0.87%	0.20%	1.10%	1.27%	0.77%	1.07%
Average	0.23%	0.13%	0.41%	0.32%	0.24%	0.27%
Median	0.05%	0.00%	0.24%	0.10%	0.06%	0.07%
Std	0.29%	0.22%	0.41%	0.39%	0.30%	0.35%

Table 39 Performance of TS-based HH-GPILS

From the proposed *VNS-based HH-GPILS*, shown in Table 41 and 42, one can conclude:

- Overall, VNS-based HH-GPILS with parameter *IM* fixed to *PIH* is performing better than the others, with average 0.16%, median 0.01% and standard deviation %0.25;
- As for computational time setting *IM* to *PIH* is the most time-consuming option.

The results of the experiments for the proposed hybrid of *SA* and *TS based HH-GPILS* are shown in Table 43 and 44. From these experiments, one can conclude:

- Overall using neighbourhood structure NS1 is not sufficient, although computationally it is more efficient than;
- Fixing IM to PIH and using only NS2 neighbourhood structure is performing better, with average 0.187%, median 0.086% and standard deviation %0.25.
- On the other hand, fixing IM to PIH and using only NS1 neighbourhood structure produces good quality solutions much faster than the other experiments.

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	53	88	35	88	50	114
eil76	271	206	84	206	38	190
pr76	83	87	71	87	48	126
kroA100	129	198	145	198	71	353
kroB100	180	251	117	251	70	312
kroC100	231	278	105	278	77	431
kroD100	131	356	94	356	49	286
kroE100	110	205	129	205	84	283
eil101	256	259	218	259	102	551
pr107	206	450	167	450	89	412
pr124	427	447	381	447	102	467
ch130	563	900	371	900	190	824
pr136	305	598	158	598	202	696
pr144	313	870	371	870	247	796
ch150	904	932	536	932	231	788
kroA150	768	875	422	875	449	724
kroB150	928	1089	533	678	351	1235
pr152	875	668	294	668	252	1007
kroA200	990	1715	1078	1715	456	2521
kroB200	1238	1043	788	1043	686	2486

Table 40 Computaional time of TS-based HH-GPILS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.05%	0.00%	0.00%
eil76	0.34%	0.48%	0.56%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.02%
kroB100	0.00%	0.00%	0.03%
kroC100	0.00%	0.00%	0.04%
kroD100	0.00%	0.09%	0.05%
kroE100	0.00%	0.00%	0.00%
eil101	0.92%	0.92%	0.60%
pr107	0.00%	0.00%	0.03%
pr124	0.00%	0.00%	0.00%
ch130	0.30%	0.40%	0.32%
pr136	0.02%	0.00%	0.03%
pr144	0.00%	0.00%	0.00%
ch150	0.42%	0.55%	0.56%
kroA150	0.24%	0.46%	0.29%
kroB150	0.02%	0.21%	0.11%
pr152	0.00%	0.09%	0.00%
kroA200	0.63%	0.70%	0.56%
kroB200	0.24%	0.84%	0.67%
Average	0.16%	0.24%	0.19%
Median	0.01%	0.05%	0.04%
Std	0.25%	0.31%	0.25%

Table 41 Performance of VNS-based HH-GPILS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	184	383	83
eil76	508	195	253
pr76	243	135	180
kroA100	409	548	353
kroB100	407	405	172
kroC100	533	558	437
kroD100	438	461	492
kroE100	372	559	318
eil101	458	530	985
pr107	680	663	302
pr124	512	600	863
ch130	613	1146	942
pr136	816	818	1019
pr144	733	863	864
ch150	878	1168	1516
kroA150	1255	936	1608
kroB150	1336	1116	1484
pr152	904	747	1210
kroA200	2058	1795	1561
kroB200	2039	1850	1856

Table 42 Computational time of VNS-based HH-GPILS

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	0.24%	0.24%	0.42%	0.05%	0.42%	0.24%
eil76	0.93%	0.41%	0.89%	0.74%	1.12%	0.22%
pr76	0.14%	0.00%	0.00%	0.00%	0.00%	0.00%
kroA100	0.02%	0.00%	0.10%	0.02%	0.01%	0.00%
kroB100	0.09%	0.00%	0.15%	0.00%	0.36%	0.00%
kroC100	0.22%	0.00%	0.23%	0.02%	0.20%	0.02%
kroD100	0.22%	0.09%	0.25%	0.13%	0.16%	0.03%
kroE100	0.01%	0.00%	0.04%	0.00%	0.19%	0.00%
eil101	1.37%	0.95%	1.81%	0.86%	1.59%	1.15%
pr107	0.04%	0.00%	0.06%	0.03%	0.07%	0.03%
pr124	0.05%	0.02%	0.05%	0.00%	0.02%	0.00%
ch130	0.60%	0.19%	0.96%	0.63%	1.02%	0.48%
pr136	0.45%	0.08%	0.06%	0.04%	0.10%	0.02%
pr144	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ch150	0.54%	0.48%	1.04%	0.55%	1.05%	0.63%
kroA150	0.64%	0.16%	0.75%	0.57%	0.42%	0.39%
kroB150	0.10%	0.10%	0.24%	0.07%	0.21%	0.09%
pr152	0.04%	0.00%	0.22%	0.00%	0.20%	0.07%
kroA200	1.06%	0.50%	1.50%	0.62%	0.58%	0.75%
kroB200	1.08%	0.53%	1.20%	1.15%	0.93%	0.84%
Average	0.39%	0.19%	0.50%	0.28%	0.43%	0.25%
Median	0.22%	0.09%	0.23%	0.05%	0.21%	0.05%
Std	0.41%	0.25%	0.54%	0.36%	0.45%	0.34%

Table 43 Performance of Hybrid of SA and TS

Instance	$IM = PIH$		$IM = AP$		$IM = \{AP, PIH\}$	
	NS1	**NS2**	**NS1**	**NS2**	**NS1**	**NS2**
eil51	30	32	142	142	57	30
eil76	45	73	384	384	54	105
pr76	37	57	191	191	100	110
kroA100	53	223	294	294	240	125
kroB100	40	170	392	392	82	231
kroC100	49	258	397	397	109	349
kroD100	90	314	460	460	134	516
kroE100	32	276	303	303	110	157
eil101	78	169	356	356	86	369
pr107	100	254	673	673	134	714
pr124	89	719	489	489	130	261
ch130	503	852	569	655	228	750
pr136	378	398	620	620	205	856
pr144	320	1077	940	940	176	383
ch150	190	693	1005	1136	442	924
kroA150	253	932	998	741	560	606
kroB150	724	748	977	874	971	266
pr152	251	304	991	893	351	422
kroA200	727	1565	1953	2548	1041	1603
kroB200	445	1107	1535	3284	638	1724

Table 44 Computational time of Hybrid of SA and TS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.00%	0.00%	0.05%
eil76	0.22%	0.37%	0.59%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.00%
kroB100	0.00%	0.00%	0.00%
kroC100	0.00%	0.04%	0.00%
kroD100	0.00%	0.08%	0.00%
kroE100	0.00%	0.00%	0.00%
eil101	0.79%	0.92%	0.57%
pr107	0.00%	0.00%	0.00%
pr124	0.00%	0.00%	0.00%
ch130	0.15%	0.54%	0.44%
pr136	0.04%	0.01%	0.00%
pr144	0.00%	0.00%	0.00%
ch150	0.35%	0.56%	0.47%
kroA150	0.17%	0.48%	0.40%
kroB150	0.04%	0.24%	0.25%
pr152	0.00%	0.07%	0.04%
kroA200	0.51%	0.69%	0.67%
kroB200	0.36%	0.77%	0.46%
Average	0.13%	0.24%	0.20%
Median	0.00%	0.06%	0.02%
Std	0.21%	0.30%	0.24%

Table 45 Performance of Hybrid of VNS and TS with shaking

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	184	383	83
eil76	508	195	253
pr76	243	135	180
kroA100	409	548	353
kroB100	407	405	172
kroC100	533	558	437
kroD100	438	461	492
kroE100	372	559	318
eil101	458	530	985
pr107	680	663	302
pr124	512	600	863
ch130	613	1146	942
pr136	816	818	1019
pr144	733	863	864
ch150	878	1168	1516
kroA150	1255	936	1608
kroB150	1336	1116	1484
pr152	904	747	1210
kroA200	2058	1795	1561
kroB200	2039	1850	1856

Table 46 Computational time of Hybrid of VNS and TS with shaking

As for the hybrid of VNS and TS based *HH-GPILS*, we experimented with the effect of shaking procedure where the current solution is perturbed by randomly changing

two random parameters of the set, see Table 45 and 46. Moreover, we experimented with the hybrid of VNS and TS without the shaking procedure, see Table 47 and 48. From the obtained results the following conclusions can be drawn:

- This *HH-GPILS* method with fixed IM to PIH is performing better, in terms of quality of the solutions, in both experiments, with (average 0.13%, median 0.0% and standard deviation 0.21%) or without shaking procedure (average 0.153% and median 0.012% and standard deviation 0.21%);
- However, the hybrid of VNS and TS with shaking procedure has better performance in terms of quality of the solutions than the one without shaking.
- As for computational time, the hybrid without shaking is performing much better than the hybrid with shaking;
- In general, the hybrid of VNS and TS without shaking procedure and with fixed IM to PIH is computationally more efficient.

Results of the hybrid SA and VNS based *HH-GPILS* is shown in Table 49 and 50. From the results, one can conclude:

- Fixing *IM* to *PIH* results in better performance, in terms of both quality of the solution and computational time, in comparison with the others, with average 0.18%, median 0.01% and standard deviation 0.28%.

Table 51 and 52 show the results for the hybrid of SA, VNS and TS based HH-GPILS. From this table one can conclude:

- By fixing *IM* to *PIH,* this hyperheuristic obtains better results, with an average of 0.18%, median 0.03% and standard deviation %0.26.
- On the other hand, fixing *IM* to *AP,* this hybrid is more efficient in terms of computational time.

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.00%	0.09%	0.09%
eil76	0.48%	0.89%	0.78%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.04%	0.02%
kroB100	0.00%	0.05%	0.03%
kroC100	0.00%	0.10%	0.08%
kroD100	0.01%	0.03%	0.03%
kroE100	0.00%	0.00%	0.00%
eil101	0.60%	1.02%	1.08%
pr107	0.01%	0.02%	0.03%
pr124	0.00%	0.00%	0.00%
ch130	0.30%	0.78%	0.71%
pr136	0.05%	0.14%	0.04%
pr144	0.00%	0.00%	0.00%
ch150	0.42%	0.93%	0.42%
kroA150	0.15%	0.48%	0.38%
kroB150	0.06%	0.20%	0.17%
pr152	0.00%	0.07%	0.04%
kroA200	0.60%	0.95%	0.73%
kroB200	0.36%	1.13%	1.30%
Average	0.15%	0.35%	0.30%
Median	0.01%	0.10%	0.06%
Std	0.21%	0.41%	0.39%

Table 47 Performance of Hybrid of VNS and TS without shaking

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	130	146	19
eil76	178	102	58
pr76	67	97	67
kroA100	202	244	160
kroB100	219	329	172
kroC100	134	517	234
kroD100	246	311	243
kroE100	199	312	212
eil101	309	550	166
pr107	267	570	302
pr124	323	620	417
ch130	651	788	463
pr136	270	598	621
pr144	394	764	385
ch150	616	772	345
kroA150	480	929	330
kroB150	583	753	504
pr152	524	357	461
kroA200	888	2643	773
kroB200	1112	2455	1394

Table 48 Computational time of Hybrid of VNS and TS without shaking

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.14%	0.00%	0.09%
eil76	0.22%	0.67%	0.30%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.00%
kroB100	0.00%	0.00%	0.00%
kroC100	0.00%	0.04%	0.04%
kroD100	0.00%	0.17%	0.00%
kroE100	0.00%	0.00%	0.00%
eil101	0.95%	1.18%	0.95%
pr107	0.00%	0.00%	0.00%
pr124	0.00%	0.00%	0.00%
ch130	0.33%	0.52%	0.27%
pr136	0.04%	0.03%	0.00%
pr144	0.00%	0.00%	0.00%
ch150	0.36%	0.47%	0.49%
kroA150	0.23%	0.29%	0.32%
kroB150	0.02%	0.22%	0.08%
pr152	0.00%	0.07%	0.04%
kroA200	0.61%	0.55%	0.62%
kroB200	0.78%	1.21%	0.62%
Average	0.18%	0.27%	0.19%
Median	0.01%	0.06%	0.04%
Std	0.28%	0.37%	0.27%

Table 49 Performance of Hybrid of SA and VNS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	28	122	66
eil76	138	229	249
pr76	248	132	155
kroA100	336	325	370
kroB100	496	548	213
kroC100	267	419	261
kroD100	632	458	292
kroE100	489	347	219
eil101	530	341	409
pr107	431	595	561
pr124	586	582	615
ch130	742	730	714
pr136	895	649	589
pr144	822	745	915
ch150	932	734	989
kroA150	1103	1450	1119
kroB150	1038	996	1075
pr152	644	1241	1097
kroA200	1336	2761	2217
kroB200	1780	1745	1731

Table 50 Computational time of Hybrid of SA and VNS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.14%	0.09%	0.09%
eil76	0.52%	0.67%	0.26%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.00%
kroB100	0.00%	0.00%	0.00%
kroC100	0.00%	0.14%	0.04%
kroD100	0.01%	0.08%	0.11%
kroE100	0.00%	0.00%	0.00%
eil101	0.95%	0.89%	1.18%
pr107	0.00%	0.00%	0.00%
pr124	0.00%	0.00%	0.00%
ch130	0.23%	0.48%	0.48%
pr136	0.07%	0.06%	0.02%
pr144	0.00%	0.00%	0.00%
ch150	0.38%	0.79%	0.56%
kroA150	0.44%	0.48%	0.39%
kroB150	0.05%	0.19%	0.15%
pr152	0.00%	0.07%	0.09%
kroA200	0.53%	0.95%	0.84%
kroB200	0.34%	1.05%	0.77%
Average	0.18%	0.30%	0.25%
Median	0.03%	0.09%	0.09%
Std	0.26%	0.36%	0.34%

Table 51 Performance of Hybrid of SA, VNS and TS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	85	31	27
eil76	296	61	110
pr76	167	65	185
kroA100	253	78	223
kroB100	333	86	262
kroC100	249	98	379
kroD100	265	102	197
kroE100	285	114	177
eil101	404	167	307
pr107	370	173	562
pr124	478	182	569
ch130	606	203	1469
pr136	453	217	440
pr144	607	269	448
ch150	614	344	373
kroA150	822	355	935
kroB150	941	401	764
pr152	720	637	551
kroA200	1407	999	1521
kroB200	2022	753	1160

Table 52 Computational time of Hybrid of SA, VNS and TS

The average and median of all proposed sequential HH-GPILS are shown in Figure 34. To conclude, from this figure and the analysis above the following conclusions can be drawn:

- Overall, sequential hyperheuristics provide good quality solutions, on average less than 0.5% increase over the optimal.
- Amongst the proposed hyperheuristics, SA-based HH-GPILS has the best performance, with a minimum overall average of 0.128% ($IM = PIH$) and a maximum overall average of 0.392% ($IM = AP$).
- In general, by fixing *IM* to *PIH* one can improve the quality of the HH-GPILS.
- Overall, neighbourhood structure NS2, in comparison with NS1, provides better quality solutions. The reason is that the diversification level of NS2 is higher than the diversification level of NS1. Thus, NS1 gets stuck in local optima more often. However, SA-based HH-GPILS, using NS1 and *IM* fixed to *PIH*, provides good quality solutions, with an average of 0.143% and a median of 0.0%. The reason is that the SA's acceptance function allows temporary acceptance of deteriorating solutions based on a probability that depends on the temperature.
- The best nine of the proposed HH-GPILS are the ones with fixed *IM* to *PIH*, and as for neighbourhood structures they all, except the fourth one SA-based HH-GPILS using $NS1$, are using neighbourhood structures $NS2$ or $\{NS1, NS2\}$.
- However, in terms of computational time, in general, setting *IM* to *AP* and using NS1 is more efficient.

4.5.3. Sequential HH-GPILS in comparison with DLS

In this section, we shall compare the proposed HH-GPILS with DLS developed by Ouenniche et al. (2017). In order to compare DLS with HH-GPILS, we classified all the results of to Ouenniche et al. (2017) in three categories depending on the choice of parameters S and r : $\{S = 2, r_1 = 1, r_2 = 1\}$, $\{S = 2, r_1 = 2, r_2 = 1\}$ and $\{S = 3, r_1 = 1, r_2 = 1 \; and \; r_3 = 1\}$. In each category, we only chose the minimum average increase over optimal amongst all. Figure 35 shows the average increase over optimal solution for the above mentioned DLS categories by Ouenniche et al. (2017), which is

shown by green bars, and all the proposed sequential HH-GPILS are shown by blue bars.

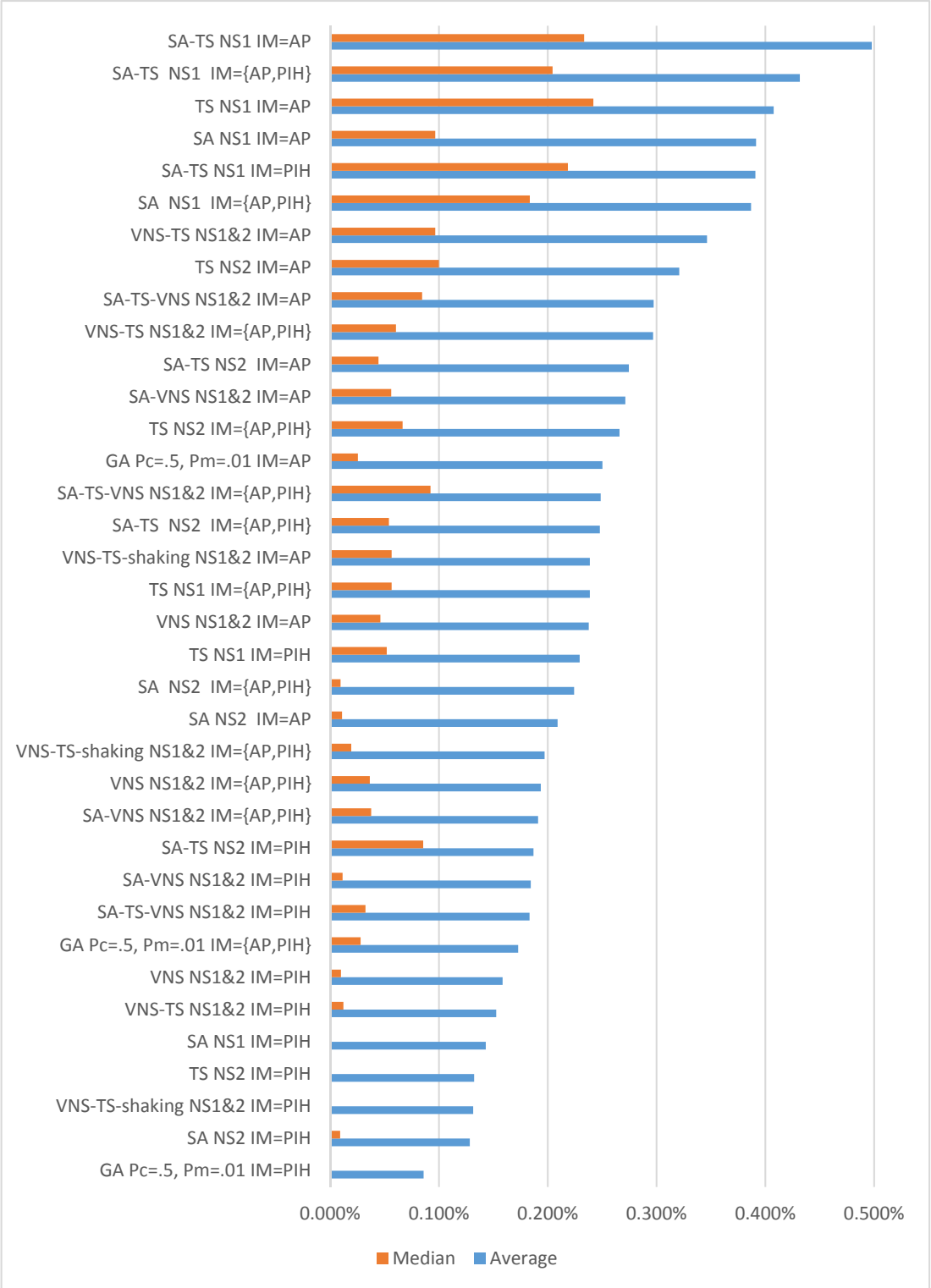


Figure 34 Average and median of all proposed sequential HH-GPILS

Recall that the set of parameters used by GPILS is not restricted as was the case with Ouenniche et al. (2017). From this figure, the following conclusions can be drawn:

- Overall, sequential HH-GPILS is performing better in comparison with DLS.
- DLS with parameters $\{S = 2, r_1 = 1, r_2 = 1\}$ is producing better solutions in comparison with parameters $\{S = 2, r_1 = 2, r_2 = 1\}$ and $\{S = 3, r_1 = 1, r_2 = 1 \; and \; r_3 = 1\}$. However, this observation does not mean that lower value of s and r perform better. The preliminary results showed that, depending on the instance of the TSP, higher values could lead to better solutions.

In general, using a hyperheuristic methods to automate the choice of parameters of the GPILS could lead to better solutions than tailor-made methods such as DLS.

4.5.4. Sequential HH-GPILS in comparison with primal methodologies

In this section a comparison between the best three sequential HH-GPILS and the relevant benchmark is presented. Table 53 and Figure 36 illustrate the results. In Table 53, the first column shows the instances of the TSP and the rest of the columns show the percentage increase over the optimal solution of each method. The best three sequential HH-GPILS are:

1. SA -based HH-GPILS (using NS2 and IM fixed to PIH) is shown in the second column as SA HH-GPILS;
2. Hybrid of VNS and TS HH-GPILS (with shaking and IM fixed to PIH) is shown in the third column as VNS-TS HH-GPILS;
3. TS-based HH-GPILS (using NS2 and IM fixed to PIH) is shown in the fourth column as TS HH-GPILS;

and the benchmark methods are as follows:

1. Metaheuristic for randomized priority search (Meta-RaPS) proposed by DePuy et al. (2005) is shown in the fifth column;
2. Adaptive TS (ATS) proposed by Suwannarongsri and Puangdownreong (2012) is shown in the sixth column;

3. Parallel Adaptive TS (PATS) approach proposed by He et al. (2005) is shown in the seventh column;
4. Adaptive SA with greedy search (ASA-GS) proposed by Geng et al. (2011) is shown in the eighth column;
5. Generalised chromosome GA (GCGA) proposed by Yang et al. (2008) is shown in the ninth column.

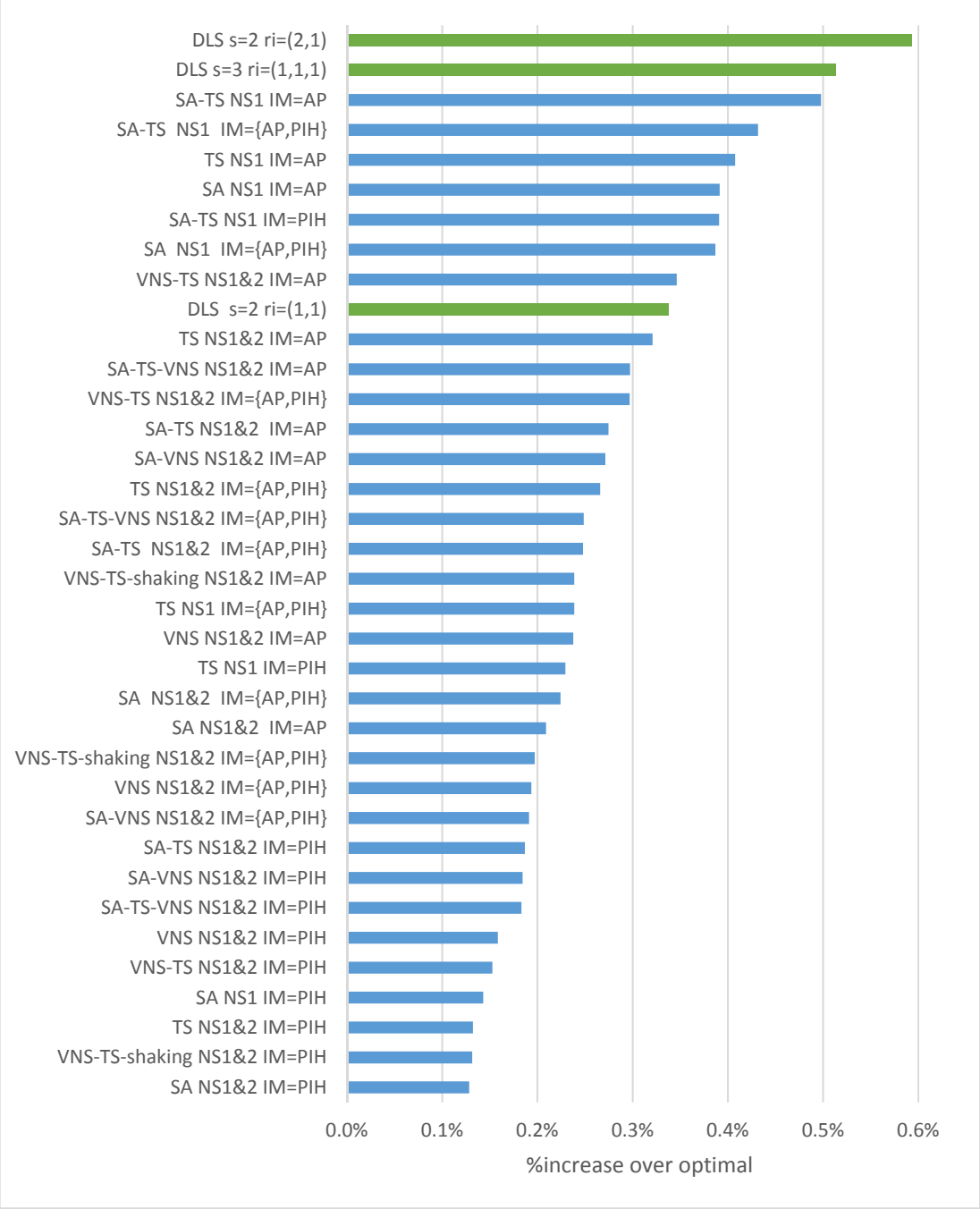


Figure 35 Sequential HH-GPILS in comparison with DLS

Instance	SA HH-GPILS	VNS-TS HH-GPILS	TS HH-GPILS	Meta-RaPS	ATS	PATS	ASA-GS	GCGA
ch130	0.19	0.15	0.20			2.44	0.18	
ch150	0.48	0.35	0.34		1.30	2.58	0.16	
eil51	0.24	0.00	0.00		2.85	1.12	0.67	0.94
eil76	0.41	0.22	0.45		0.40	2.71	1.18	2.42
eil101	0.95	0.79	0.83			2.71	1.83	2.70
kroA100	0.00	0.00	0.00	0.00	1.14	0.37	0.01	1.23
kroA150	0.16	0.17	0.16				0.05	2.92
kroA200	0.50	0.51	0.42	1.07		2.62	0.23	1.85
kroB100	0.00	0.00	0.00	0.25	1.93	1.78	0.00	1.81
kroB150	0.10	0.04	0.02				0.18	2.11
kroB200	0.53	0.36	0.20	1.26			0.25	4.04
kroC100	0.00	0.00	0.00	0.00		1.37	0.00	1.33
kroD100	0.09	0.00	0.00	0.00		4.72	0.03	2.42
kroE100	0.00	0.00	0.00	0.17			0.20	1.41
pr76	0.00	0.00	0.00		2.14			
pr107	0.00	0.00	0.00	0.00		0.35	0.00	1.37
pr124	0.02	0.00	0.00	0.00		0.77	0.00	0.19
pr136	0.08	0.04	0.04	0.39		1.05	0.31	2.82
pr144	0.00	0.00	0.00				0.01	0.04
pr152	0.00	0.00	0.00	0.00			0.01	1.22

Table 53 Sequential HH-GPILS in comparison with primal methodologies

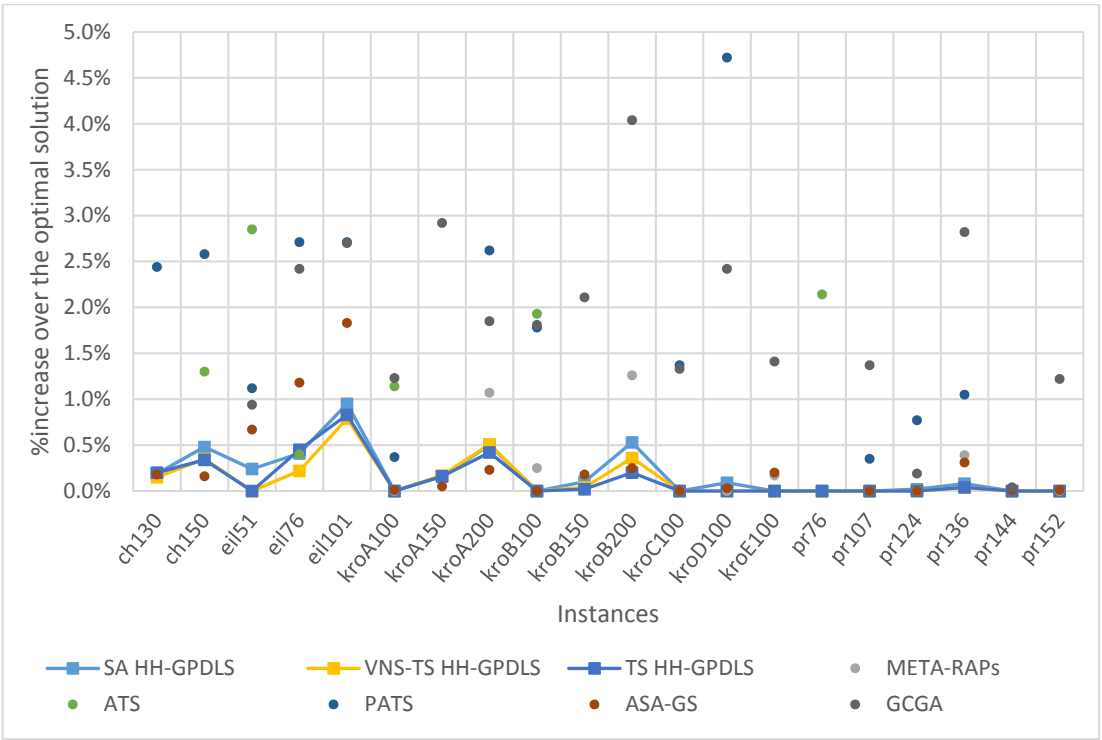


Figure 36 Sequential HH-GPILS in comparison with primal methodologies

In Figure 36, the vertical axis shows the average percentage increase over the optimal and the horizontal axis shows TSP instances. Each line in the figure represents the quality of a sequential HH-GPILS, above mentioned, and the markers show each of the benchmarks. Overall, the proposed sequential HH-GPILS outperforms the benchmark, with some few exceptions. ASA-GS is performing better for instances ch150, kroa150 and kroa200, however, on average the proposed sequential HH-GPILS are performing better than ASA-GS.

4.6. Conclusion

Sequential hyperheuristics are smart techniques that select the best set of parameters for GPILS, each using a different search strategy and neighbourhood structures. Overall, they all provide good quality solutions, on average less than 0.5% increase over the optimal and deliver the optimal solution in 55% of experiments conducted. In comparison with the benchmark, DLS, the proposed HH-GPILS has a better performance because of its adaptability.

However, the neighbourhood structure and the bounds of the set of parameters has a significant influence on the quality of the solution. A neighbourhood structure which provides the right level of intensification and diversification could lead to the best path across the search space and lead to the global optimal solution.

As for the bounds of the parameters, in this study, we mostly concentrated on the initialisation of the infeasible solution. In general, initialising the infeasible solution using the *PIH* provides better quality solutions than AP, however using the AP to initialise the infeasible solution is computaionalyy more efficient.

Finally, the proposed GPILS can be categorised as constructive-perturbative because of its infeasible nature and the use of Type II move to improve the infeasible components. Its performance is clearly enhanced when implemented within a hyperheuristic framework.

5. A Parallel Hyperheuristic Framework for GPILS

In the previous chapter we developed sequential hyperheuristics to automate and optimise the choice of the set of parameters of GPILS. These hyperheuristics start with single set of parameters and search its neighbourhood for a better set. On the other hand, one can start with a population of sets of parameters. In this chapter, we propose a parallel or population-based hyperheuristic for optimising the set of parameters of GPILS.

As it was mentioned in section 2.5.2, traditional parallel algorithms, such as genetic algorithm (GA) and memetic algorithm (MA), start with a population of individuals (chromosomes), representing the solution to the problem under consideration, and evolves them to create new population hoping that these individuals have better performance while inheriting their parents' features.

Furthermore, the quality of the initial population can have a significant influence on the performance of the GA and its convergence speed. In other words, starting with a good-quality population can speed up the search, although, it might prematurely converge to a local optimum. Initialising the population using seeding techniques can generate high-quality chromosomes can improve the quality, and can speed up, the GA's search for the best solution. Although, these techniques might increase the chance of immature convergence. There are several seeding techniques to initialise the population such as initialising the population using a heuristic (Yang, 1997; Liao, 2009; Ray et al.; 2007; Kaur and Murugappan, 2008), using a gene bank (Wei et al. 2007), sorted population (Yugay et al., 2008), etc. based on the analysis done by Paul et al. (2015) and Shanmugam et al. (2013).

In this chapter, we propose a population-based framework, namely GA-based hyperheuristic that makes use of indirect presentation of chromosomes where each chromosome represents a set of parameters of GPILS, see Figure 37. The proposed population-based hyperheuristic evolves these chromosomes to find better chromosomes. Note that, to the best of our knowledge, no previous attempt has been made to optimise the parameters of a heuristic using a hyperheuristic framework.

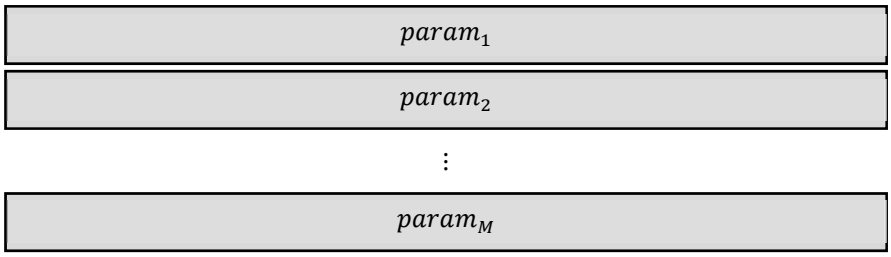


Figure 37 Population of parallel hyperheuristic

Later, we proposed an offline learning mechanism for GA-based HH-GPILS, with aim of reusing the set of parameters on unseen (new) problem instances after such set of parameters has evolved on a given set of training problem instances. The proposed offline learning mechanism makes use of a knowledge-based system that keeps track of good performing chromosomes, i.e. set of parameters of the GPILS. The knowledge-based system is refered to as chromosomes bank (CB) that keeps several good performing chromosomes along with their scores, see Figure 38. When solving a new TSP instance, all or a set of these chromosomes are used to initialise the population of the GA. Note that the score of the set of chromosomes used to initialise the GA is updated based on their performance. When GA converges to the final population or stops the search before the convergence occurs, all or a number of the best performing chromosomes of the final generation will replace a set of existing chromosomes in the CB.

In summary, the contributions of this chapter are developing a population-based hyperheuristic to automate and optimise the choice of parameters of GPILS and proposing an offline learning mechanism for GA-based HH-GPILS. Hereafter, we shall discuss the implementation decisions of the population-based methods. We divided these implementation decisions to problem-specific and generic decisions. Problem specific decisions are common decisions in the implementation of

population-based methods. However, generic decisions are dependent on the choice of the high-level methodology. Afterwards, the details of the proposed offline learning are explained. Later, the empirical investigation and finally conclusions are presented.

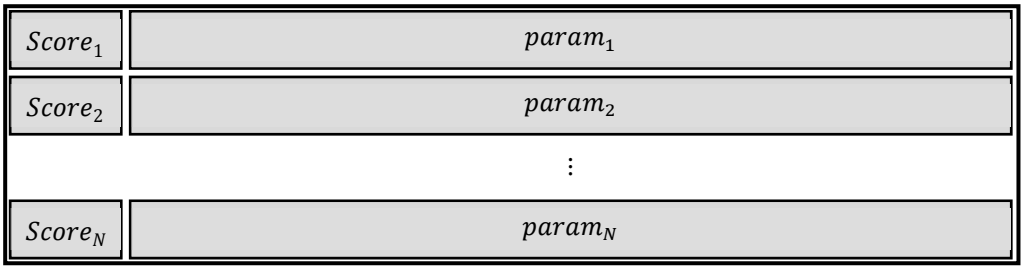


Figure 38 Chromosomes bank

5.1. Problem-specific decisions for high-level search mechanisms

These decisions are common amongst population-based methods but dependent on the problem under consideration. In the concept of HH-GPILS, these decisions are related to the GPILS. Problem-specific to the implementation of population-based hyperheuristics is (1) choice of the parameters' space; (2) choice of the genetic representation or encoding scheme of chromosomes; and (3) choice of the fitness measure. We shall discuss these decisions in the next section

Choice of the parameters' space: The choice of the parameters' space is similar to the choice of the parameters' space used for sequential hyperheuristics.

Genetic representation or encoding scheme of chromosomes: Since chromosomes are a string of genes representing its genetic information, in our population-based hyperheuristics; a chromosome is represented as a vector (string) of parameters of GPILS.

Fitness measure: The choice of fitness measure is similar to the choice of the form of the optimisation function used for sequential hyperheuristics.

5.2. Choice of the high-level methodology and its implementation decisions

As it was mentioned in the previous chapter, the high-level methodology for searching the parameter space of GPILS could either be a sequential methodology or a parallel one. In this chapter, we opt for experimenting with parallel (population-based) methodologies. To be more specific, we have chosen Genetic Algorithm (GA) as a high-level methodology. Since the problem-specific decisions are discussed earlier, in the next section, we shall discuss generic implementation decisions of the proposed metaheuristics for searching the parameter space of GPILS.

5.2.1. Genetic algorithm as a high-level search mechanism

Genetic algorithms (GAs) are population-based metaheuristics that start the search with an initial population and evolves those using bio-inspired operators to generate offsprings hoping that the new generation inherits the parent's good genes and be better than their parents. The customised pseudo-code of the GA as high-level methodology algorithm is outlined in Table 54.

As it was mentioned in earlier, designing a GA requires two sets of decisions, namely problem-specific decisions and generic decisions. Since in this study GA is used as a high-level methodology to search the heuristic space the problem-specific decisions should be modified. However, the generic decisions for GA-based hyperheuristic could be similar to the operators in traditional GA, note that some or all of them need slight modifications. Thus, we customised the GA problem-specific and generic decisions to our search in the search space of GPILS.

Generic decisions are concerned with the parameters of the algorithm itself. The generic decisions are choice of (1) population size and selection of initial population; (2) parents' selection and replacement mechanism; (3) reproduction mechanism; (4) immigration operator; (5) genetic operators' rates and (6) stopping criteria. We shall discuss these decisions in the next section.

Population size and selection of initial population: Since the choice of population size M has a great influence on GA's efficiency and effectiveness, one should make a trade-

off when choosing the population size. In order to initialise the population, one can choose to randomly generate M chromosomes, however one might initialise the population based on historical knowledge about the TSP instance and/or GPILS performance (e.g. offline learning or using a trial run). In our empirical investigation, we have experimented with random seeding.

Initialisation Step

Choose an initial population of M individuals of GPILS, $param_m$, in the admissible parameter space S evaluate the fitness of each individual, $z(param_m)$; that is, the total distance of the TSP tour constructed by GPILS using the set of parameters of individual $param_m$;

Initialise the best solution found so far, say $(param^*, z^*)$, by setting $param^* = param_0$ and $z^* = z(param_0)$;

Set iteration counter I to 0;

Set Best-Found-At-iteration to 0;

Set immigration counter I_{imm} to 0;

Iterative Step

REPEAT until stopping condition = true

IF crossover condition(s) hold **THEN** {

Select a subset of individuals from the current generation as parents for reproduction;

Perform a crossover operation on parents to generate children; }

IF mutation condition(s) hold **THEN** {

Select a subset of individuals from the current generation as parents for reproduction;

Perform a mutation operation on parents to generate children; }

IF immigration condition(s) hold **THEN {**

Perform an immigration operation to generate children;

Increment immigration counter by 1; that is, set $I_{imm} = I_{imm} + 1$;}

Evaluate the fitness of each child and update the best solution found so far, if necessary;

IF $z(param') < z^*$ **THEN {**

update the best vector of parameters found so far; that is, set $param^* = param'$ and $z^* = z(param')$;

Best-Found-At-iteration = I;}

Replace a subset of parents in the current population by a subset of the current children to produce a new generation;

Increment iteration counter by 1; that is, set $I = I + 1$;

END REPEAT

Table 54 Pseudo-code of genetic algorithm as a high-level methodology

Parents' selection and replacement mechanism: GA iteratively selects some chromosomes, from the current generation, for mating and combines them to generate new offsprings; these new offsprings replace a number of chromosomes, from the current generation, to produce a new generation. In our empirical investigation, we used steady-state selection mechanism where two subsets with equal sizes are chosen, the first subset for mating and the second to be deleted. In order to choose parents for the mating pool, we experimented with tournament selection ($t = 2$). On the other hand, to select chromosomes to be replaced we experimented with deleting the worst individual from the current generation, with consideration of replacement-with-no-duplicates.

Reproduction mechanism: Reproduction mechanism consists of two operators. The first operator, crossover, combines the parents by swapping some alleles to produce offsprings. The second operator, mutation, diversifies the chromosome by randomly introducing new features into the chromosomes. In our implementation for the crossover operation, we experimented with uniform crossover (Syswerda, 1989; Spears and De Jong, 1995). In uniform crossover mechanism, parents' alleles are randomly swapped with probability p_{uc} ($p_{uc} = 0.5$). Using this crossover mechanism produces new offsprings, while these new offsprings are inheriting the parents' genes and information with higher level of diversification in comparison with one-point and two-point crossovers' mechanisms.

As for the mutation operator, we experimented with random mutation of chromosomes, where a number of alleles are changed randomly. The choice of alleles to change is made using the mutation rate.

Genetic operators' rates: The rates of the genetic operators control the evolution of the current population. Crossover and mutation rates specify the rate each of these operations is used. The first, crossover rate, indicate how many offsprings are introduced into the population by combining their parents whereas the second, mutation rate, indicates the rate where new information is entered in the population.

Immigration operator: Immigration operator introduces new individuals to replace a proportion of existing individuals, typically the worst individuals. This operator is used to add a level of diversification to the GA. An effective immigration operator allows

the GA to explore different regions of the search space. In our empirical investigation, we randomly generated a number of new individuals ($Imm = {}^{M}/_{2}$). These new individuals are used to replace Imm number of worst performing individuals in the current generation. The empirical investigation showed that the appropriate condition for immigration to only when either of the following conditions occur:

1. If the population converges and immigration has not occurred for four iterations.

   The reason for this condition is that to avoid premature convergence. Moreover, the occurrence of immigration has been restricted to a number of iterations to prevent the unnecessary random search.

2. If no immigration has occurred but the iteration counter I reached the maximum number of evolutions and the percentage population convergence is greater than predefined percentage.
   The reason is that assuming the initial population did not lead to the optimal solution and the search got stuck in the local optima; we use immigration operator to get out of the local optima. However, immigration might not be necessary, if this was not the case.
3. If a better solution has not been found for some iterations and immigration has not been occurred for four iterations.
   When this condition happens, one might say, again, that the search got stuck in the local optima and immigration operator could be used get out of it.

As it was mentioned, the occurrence of the immigration operator should be restricted to avoid the unnecessary random search. Thus, we limited immigration occurrence for a number of iterations. Also, immigration operator can be used for maximum three times.

Stopping criteria: Several stopping criteria can be utilized in the GA-based high-level mechanisms such as predetermined number of evolutions or generations was reached, maximum number of iteration since the last best individual was found, maximum number of iteration since the last best individual was found, the highest number of immigrations has occurred, measure of the population diversity fell below a

prespecified threshold. We experimented with several stopping criteria, the appropriate stopping criteria were the first three criteria, when either of these criteria occurs the GA's search for better individual stops.

5.3. Parallel Hyperheuristic Framework with Offline Learning Mechanism for GPILS

As it was mentioned in the previous section, GA starts with an initial population. The quality of the initial population has a significant influence on the performance of the GA and its convergence speed. In other words, starting with a high-quality population can speed up the search. However, GA might converge prematurely to a local optimum. Seeding techniques that generate high-quality chromosomes can improve the quality, and can speed up, the GA's search for the best solution. Although, these techniques might increase the chance of immature convergence. There are several seeding techniques to initialise the population such as initialising the population using a heuristic (Yang, 1997; Liao, 2009; Ray et al.; 2007; Kaur and Murugappan, 2008), using a gene bank (Wei et al. 2007), sorted population (Yugay et al., 2008), etc. based on the analysis done by Paul et al. (2015) and Shanmugam et al. (2013).

In this section, we proposed an offline learning mechanism for GA-based HH-GPILS, which makes use of a knowledge-based system keeping track of good performing chromosomes that is set of parameters of the GPILS. The knowledge base system is refered to as chromosomes bank (CB) that keeps several good performing chromosomes along with their scores. When solving a new TSP instance, all or a set of these chromosomes are used to initialise the population of the GA. Note that the score of the set of chromosomes used to initialise the GA will be updated based on their performance. When GA converges to the final population or stops the search before the convergence occurs, all or a number of the best performing chromosomes of the final generation will replace a set of existing chromosomes in the CB.

Hereafter, the details of the proposed offline learning is explained; namely, the initialisation of the CB for the first time, using CB to initialise the population in GA-based HH-GPILS, CB's score allocation and updating CB.

5.3.1. Initialising chromosomes bank (CB)

The proposed knowledge-based system, referred to as chromosomes bank (CB), keeps track of good performing chromosomes in order to reuse and adapt them to new (unseen) problem instances. Before using the CB one has to initialise the CB. In order to initialise the CB, we run GA-based HH-GPILS, for several times, to solve a number of TSP problems in the training set. In this step, the GA starts with a random initial population. All the chromosomes in the last generations, of all trial runs, are sorted in a non-decreasing order, and the best performing chromosomes of the last generations are saved in the CB, with a score equal to one.

5.3.2. Initialising the population using CB

Assuming the best set of chromosomes found so far will overall perform well for the unseen problems, when facing a new TSP problem to solve, a number of chromosomes from the CB are retrieved using a selection mechanism to initialise the population. The number of chromosomes to retrieve from the CB could be equal to the size the initial population of GA-HH, say M. As for the selection mechanisms, one might use one of the traditional selection mechanisms used in GA; namely random selection, tournament selection or roulette wheel, see Appendix E. However, another might use seeding techniques such as sorted population. Note that random selection mechanism does not depend on the chromosome's fitness or score. Conversely, the rest of the selection mechanisms do. In other words, one might use chromosome's score as the selection criterion; another might use their fitness which is obtained by performing GPILS for each chromosome. In our investigation, we used sorted population as selection mechanism and fitness as a selection criterion.

5.3.3. Score allocation

We proposed a reward-based mechanism similar to reinforcement learning (RL) where feedback is provided, regarding reward and penalty, based on the overall

chromosome's performance. RL is an online learning mechanism that rewards an improving low-level heuristic (LLH) by increasing its score; otherwise, it will penalise the LLH by decreasing its score, see Appendix I. However, since our learning mechanism is offline, we proposed a different reward mechanism, where the performance of each chromosome in the CB and all the new chromosomes are calculated and their reward or penalty is allocated based on their performance. Performance of a chromosome in the CB, say CB_i, is defined as the percentage increase of each CB_i over the worst performing chromosome in the CB, say CB_{Worst}:

$$P(CB_i) = \frac{CB_{Worst} - \overline{Fitness}(CB_i)}{\overline{Fitness}(CB_i)} \times 100 \quad 35$$

where $\overline{Fitness}(CB_i)$ is calculated as the average fitness of chromosome CB_i performed to solve the new problem. In order to decide whether a chromosome is rewarded or penalised, first the average performance, say $\bar{P}$, of all chromosomes CB_i is calculated, then the weight of each CB_i is defined as the increase or decrease from $\bar{P}$ as follows:

$$weight(CB_i) = P(CB_i) - \bar{P} \quad 36$$

Chromosome CB_i is rewarded if its weight is non-negative; otherwise, it is penalised. Thus, the score of each chromosome CB_i is updated, by the proposed reward and penalty schemes, as follows:

Reward scheme:

$$Score(CB_i) = score(CB_i) + \sqrt{weight(CB_i)} \quad if\ weight(CB_i) \geq 0 \quad 37$$

Penalty scheme:

$$Score(CB_i) = Score(CB_i) - \sqrt{|weight(CB_i)|} \quad if\ weight(CB_i) < 0 \quad 38$$

As for new chromosomes (NC_j), their performance and, consecutively, their weight is computed similar to the chromosomes in the CB. In other words, their performance is defined as the percentage increase over the worst performing chromosome from the CB (CB_{Worst}) and their weight is defined as the increase from $\overline{P}$, average performance of chromosomes from CB.

$$P(NC_j) = \frac{CB_{Worst} - Fitness(NC_j)}{Fitness(NC_j)} \times 100 \quad (39)$$

$$weight(NC_j) = \bar{P} + P(NC_j) \quad (40)$$

Since their weight is always positive, their score is updated using the aforementioned reward scheme.

5.3.4. Updating chromosomes Base

The CB needs to be updated after solving a new problem. In other words, the score of the existing chromosome should be updated, depending on their performance. Moreover, some of the existing chromosomes in the CB should be replaced by the new chromosome. The criteria for replacement could be similar to the ones used in GA's replacement mechanism.

One can choose to simply add all new generation into the CB, without deleting any of the previous chromosomes from the CB, thus, expanding the set of chromosomes and possibly improving the knowledge of the learning system. However, this decision will increase the size of the CB and therefore will increase the retrieval time of chromosomes from the CB. Moreover, some of the saved chromosomes might not perform well on all new cases, and consequently, reduce the performance of the GA. Thus, we proposed replacing a subset of chromosomes in the CB by new ones, obtained from the final population of the GA's search for the optimal or near optimal set of chromosomes to solve the new problem at hand.

In order to update the CB, first, one should decide how many,$NB_{replace}^{CB}$, and which chromosomes should be replaced. In our experiments, we replaced a small percentage, say ε, of the chromosomes in the CB by the new ones.

$$NB_{replace}^{CB} = min\{\, \varepsilon \times |CB|,\ |NC| \,\} \quad (41)$$

where $|CB|$ is the size of the CB and $|NC|$ is the number of new chromosomes. In the empirical analysis we investigated several values for ε.

As for the replacement criteria, the chromosomes are first sorted based on their performance and only half of the worst performing chromosomes are considered in the

replacement process. Later, the chosen set, for the replacement process, is sorted based on their score in a non-decreasing order and only the first $NB_{replace}^{CB}$ chromosomes from the sorted list will be removed from the CB.

After eliminating a number of chromosomes from the CB, a set of new chromosomes the same size as the ones removed from CB ($NB_{insert}^{NC} = NB_{replace}^{CB}$) will be inserted into the CB. All the new chromosomes are sorted based on their score, in a non-increasing order. From the top of the list, NB_{insert}^{NC} will be inserted into the CB, without duplication. However, if the new chromosome, say NC_j, to add is already in the list, say CB_e, its score will be updated as follows:

$$Score(CB_e) = score(CB_e) + score(NC_j) \quad 42$$

5.4. Empirical investigation

As was mentioned before, in this chapter we experimented with GA-based HH-GPILS methodology. We divided the implementation decisions of this hyperheuristic to problem-specific and generic decisions. These decisions are shown in Table 55.

Type of Decisions	Decision
Problem-specific	Parameters' space
	Genetic representation or encoding scheme of chromosomes
	Fitness measure
Generic	Population size and selection of initial population
	Reproduction mechanism
	Genetic operators' rates
	genetic operators' rates
	Stopping criteria

Table 55 GA- based HH-GPILS high-level decisions

We also proposed an offline learning for the GA-based HH-GPILS methodology. The chromosomes saved in the proposed CB used in offline learning are used to initialise the population of the GA-based HH-GPILS. The empirical investigation shows that this learning mechanism can produce solutions with similar quality produced by the

GA-based HH-GPILS without learning, however, it reduces the time approximately by one third.

5.4.1. Experimental setup

For the choice of the parameters space of GPILS and the fitness measure in this empirical investigation, is similar to the ones chosen for sequential-based HH-GPILS. As for the generic decisions, they are presented in the description of each decision. However, their parameters are presented hereafter:

- Population size M is set to 40;
- Crossover probability P_c is set to 0.5;
- Probability of uniform crossover p_{uc} is set to 0.5;
- Tournament size is set to two individuals at a time;
- Mutation probability P_m is set to 0.01;
- Maximum number of evolutions is equal to 50;
- Maximum number of immigrations is set to three;
- Maximum number of iterations since the last best individual found is set to 20.

In order to test the influence of initialising the infeasible solution using either AP or *PIH* for the proposed GA-based HH-GPILS methods, first, we experimented with fixing *IM* to either AP or *PIH*. Similar to findings of the previous chapter, initialising the *IM* using *PIH* produces better quality solutions. In general, the empirical results shows that this HH-GPILS is performing better than the sequential ones.

As for the generic decisions of the GA-based HH-GPILS with offline learning, they are the set to the values mentioned above, except for the maximum number of iterations since the last best individual found. The empirical investigation showed that, in trade-off between time and quality, the most appropriate number of iterations is 15. As for the configuration of the offline learning, its parameters are as follows.

- Maximum number of chromosomes in the CB is set to 100.
- Training instances are *eil51* and *pr76*.
- Stopping time set to $\frac{n}{100} \times 600$

5.4.2. Experimental results GA-based HH-GPILS

The statistics represented are the average percentage increase over the optimal solution for a number of runs, in this investigation 5 runs. In Table 56, the first column shows the instances solved, as for the rest of the columns, each column shows the results of the following experiments:

1. First experiment: parameter *IM* is fixed to AP;
2. Second experiment: parameter *IM* is fixed to AP;
3. Third experiment: parameter *IM* is not fixed.

Hereafter, we shall discuss the performance of the proposed hyperheuristic.

Results of GA-based *HH-GPILS* are shown in Table 56 and 57. From the results, one can draw the same conclusions as the previous chapter on the sequential HH-GPILS, such as:

- Overall, GA-based HH-GPILS is performing well and is producing good quality solutions, although it is more time consuming than the sequential HH-GPILS.
- Fixing *IM* to *PIH* results in better performance in comparison with the others, with average 0.09%, median 0.0% and standard deviation 0.17%.
- On the other hand, fixing *IM* to AP is the worst performing, with average 0.25%, median 0.03% and standard deviation 0.32%, although it is computationally more efficient.

5.4.3. Parallel HH-GPILS in comparison with sequential HH-GPILS

Further, we compared the performance of the GA-based HH-GPILS with the performance of a number of the twenty of the best performing sequential HH-GPILS in the previous chapter, see Figure 39. To conclude, from this figure the following conclusions can be drawn:

- Overall, the parallel HH-GPILS provides good quality solutions.
- In general, by fixing *IM* to *PIH* one can improve the quality of the HH-GPILS.

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	0.00%	0.00%	0.00%
eil76	0.11%	0.60%	0.26%
pr76	0.00%	0.00%	0.00%
kroA100	0.00%	0.00%	0.00%
kroB100	0.00%	0.00%	0.00%
kroC100	0.00%	0.08%	0.02%
kroD100	0.00%	0.01%	0.15%
kroE100	0.00%	0.00%	0.00%
eil101	0.57%	0.92%	0.60%
pr107	0.00%	0.00%	0.00%
pr124	0.00%	0.00%	0.00%
ch130	0.01%	0.65%	0.21%
pr136	0.00%	0.00%	0.01%
pr144	0.00%	0.00%	0.00%
ch150	0.25%	0.54%	0.52%
kroA150	0.11%	0.36%	0.30%
kroB150	0.01%	0.22%	0.04%
pr152	0.00%	0.04%	0.04%
kroA200	0.52%	0.72%	0.69%
kroB200	0.14%	0.86%	0.62%
Average	0.09%	0.25%	0.17%
Median	0.00%	0.03%	0.03%
Std	0.17%	0.32%	0.24%

Table 56 Performance of GA-based HH-GPILS

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil51	572	117	130
eil76	629	353	502
pr76	743	296	388
kroA100	939	423	788
kroB100	811	331	608
kroC100	1696	410	664
kroD100	751	674	843
kroE100	824	437	705
eil101	867	554	892
pr107	904	1160	1499
pr124	1936	860	1072
ch130	2378	698	983
pr136	2538	2323	1363
pr144	2609	1532	981
ch150	2783	1003	1767
kroA150	2851	1457	1662
kroB150	2774	1242	1554
pr152	2832	1187	1438
kroA200	3878	3365	2017
kroB200	3697	3038	3108

Table 57 Computational time of GA-based HH-GPILS

- GA-based HH-GPILS with *IM* fixed to *PIH* provides the best quality solutions (average 0.09%, median 0.0% and standard deviation 0.17%).

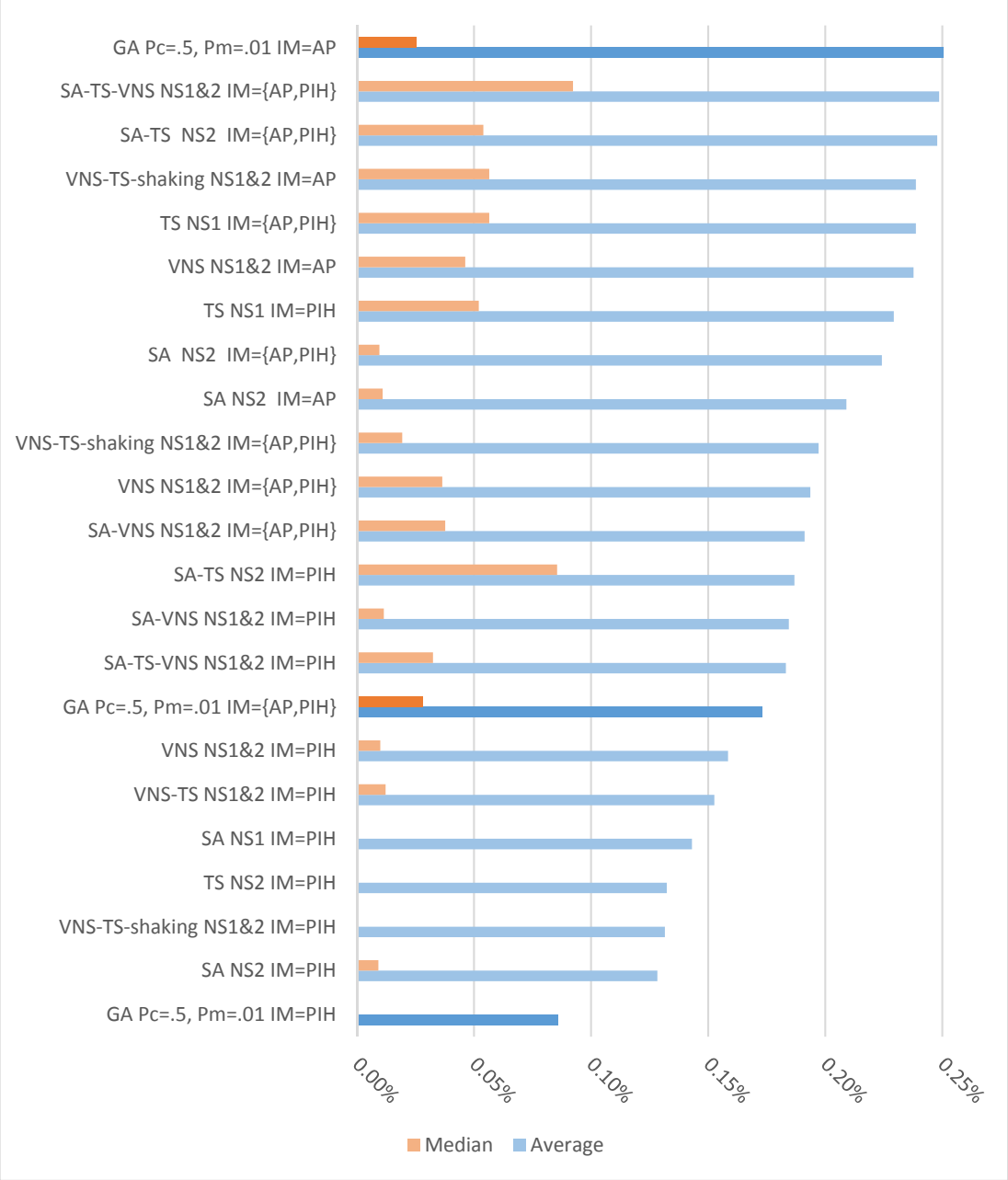


Figure 39 Parallel HH-GPILS in comaprison with sequential HH-GPILS

5.4.4. Experimental results of the GA-based HH-GPILS with offline learning

Hereafter, we shall discuss the performance of the proposed hyperheuristic. First, we experimented the reusability of the chromosomes saved in the CB from old

experiences of the GA on previous problems. Table 58 shows the performance of the chromosomes saved in the CB for the new problem at hand, before applying the HH-GPILS. The statistics represented are the average percentage increase, in cost of the best performing chromosome in the CB performed on a new problem, over the optimal solution for a number of runs, in this investigation of 5 runs. From the presented results one can conclude that:

- The offline learning could produce good quality solutions for new problem instances in a shorter time, less than 30 seconds.
- The saved sets of parameters obtained from earlier investigations could be used to address other problem instances (they are usable sets).

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil76	1.26%	2.23%	1.45%
kroA100	0.01%	0.00%	0.00%
kroB100	0.00%	0.29%	0.05%
kroC100	0.00%	0.10%	0.00%
kroD100	0.22%	0.58%	0.24%
kroE100	0.00%	0.00%	0.00%
eil101	1.30%	2.07%	1.53%
pr107	0.02%	0.00%	0.05%
pr124	0.03%	0.08%	0.00%
ch130	0.54%	1.31%	0.93%
pr136	0.26%	0.09%	0.09%
pr144	0.00%	0.00%	0.00%
ch150	0.66%	1.44%	1.19%
kroA150	0.62%	0.88%	0.99%
kroB150	0.63%	0.46%	0.20%
pr152	0.00%	0.46%	0.19%
kroA200	1.29%	0.93%	1.77%
kroB200	1.33%	1.03%	1.05%
Average	0.46%	0.66%	0.54%
Median	0.24%	0.46%	0.19%
Std	0.51%	0.69%	0.62%

Table 58 Performance of the offline learning

Results of GA-based HH-GPILS with offline learning are shown in Table 59 AND 60. From the results, one can conclude:

- However, GA-based HH-GPILS with offline learning and *IM* fixed to *PIH*, provides the best quality solutions, average 0.23%, median 0.11% and standard deviation 0.29%.

- Like the previous chapter initialising the infeasible solution with *PIH* lead to better quality solutions.
- The use of offline learning to initialise the population could produce good quality solutions for new problems in a limited shorter time.
- Overall, the shown GA-based HH-GPILS with offline learning provides good quality solutions faster.

Instance	$IM = PIH$	$IM = AP$	$IM = \{AP, PIH\}$
eil76	0.22%	0.67%	0.37%
kroA100	0.00%	0.00%	0.00%
kroB100	0.00%	0.05%	0.00%
kroC100	0.13%	0.02%	0.00%
kroD100	0.00%	0.05%	0.03%
kroE100	0.00%	0.00%	0.00%
eil101	0.73%	1.02%	1.27%
pr107	0.00%	0.00%	0.01%
pr124	0.02%	0.00%	0.00%
ch130	0.46%	0.53%	0.19%
pr136	0.17%	0.01%	0.02%
pr144	0.00%	0.00%	0.00%
ch150	0.36%	0.68%	0.80%
kroA150	0.41%	0.61%	0.74%
kroB150	0.09%	0.14%	0.08%
pr152	0.00%	0.13%	0.04%
kroA200	0.56%	0.70%	1.08%
kroB200	0.99%	0.72%	0.78%
Average	0.23%	0.30%	0.30%
Median	0.11%	0.09%	0.03%
Std	0.29%	0.34%	0.42%

Table 59 GA-based HH-GPILS with offline learning

5.5. Conclusion

Genetic algorithm is an adaptive and robust technique that requires a minimum domain-specific knowledge. Genetic algorithm's main feature is that it starts with an initial population and iteratively evolves the population, keeping most of the gathered information about the system, and produces a better population. In this chapter, we proposed a GA-based hyperheuristic to automate the choice of the parameters of GPILS.

The obtained results in this chapter showed that the proposed population-based HH-GPILS performs better than the sequential ones, however it is more time consuming, see Figure 39. As for the effect of the initialisation of the infeasible solution, the same conclusion from the last chapter can be drawn, initialising the infeasible solution using the *PIH* could lead to better solutions than the AP. On the other hand, initialising the infeasible solution using the *AP c*omputationally more efficient, since the parameters of the *PIH* are not considered in the parameters space.

Moreover, the initial population has great influence on the performance and/or speed of convergence of the GA-based HH-GPILS. Since the set of parameters of the GPILS can be reused to be adapted on new problems, one can develop knowledge-based system to keep track of the good performing sets, i.e. chromosomes, and allocate a score to each set in order to initialise the GA population using a seeding technique. In this chapter, we proposed an offline learning that uses a knowledge-based system, referred to as CB, to keep track of the best performing chromosomes, i.e. sets of parameters of GPILS, and their allocated scores. These scores are updated by reward or penalty depending on their performance in comparison with others and used to decide whether they will be replaced by new chromosomes or not. The obtained results show that this seeding technique could produce good quality initial population which lead to obtaining good optimal or near-optimal solutions by the proposed GA-based HH-GPILS more quickly.

6. Conclusion

In this section we shall present the summary and conclusions of this study as well as future research directions and final remarks.

6.1. Summary and Conclusion

In this thesis, an overview of the most common and relevant literature was presented and the heuristic solution approaches were classified into three categories; namely, feasible (primal), infeasible methodologies, and feasible-infeasible. Most of the literature on heuristic solution approaches are in the first two categories, i.e. feasible (primal), feasible-infeasible, except the work done by Ouenniche et al (2017). Ouenniche et al (2017) made the first attempt to solve the TSP by only exploring its infeasible space. This thesis refines and extends Ouenniche et al (2017) proposed infeasible search framework. This research started with the aim of investigating the potential of exploring the infeasible space for solving COPs to optimality. We first proposed a generic and parameterized infeasible local search (GPILS) as a refinement of the DLS framework proposed by Ouenniche et al (2017), where we customised GPILS to solve the TSP. The proposed GPILS starts with an infeasible solution and explores the infeasible space by repairing the current infeasible solution and reducing the infeasibilities. The proposed GPILS can be categorised as constructive-perturbative heuristic framework, constructive because of its infeasible nature and perturbative because of the use of Type II move to improve the infeasible components. The proposed refinements consists of proposing an alternative and more appropriate method to initialise infeasible local search. In addition, a recursive function was proposed to allow for the automation of the implementation of infeasible search under any set of parameters as compared to the original DLS where there was a need to write a different code to accommodate each set of parameters. Furthermore, a generic patching procedure was proposed as a generalisation of the one proposed initially by

Ouenniche et al (2017). Last, but not least, the enhanced version allows for primal search to be performed after the infeasible search, if needed.

Since the GPILS is a parameterised method, it could be seen as a collection of infeasible search methods, where its sets of parameters could be chosen by the analyst or an automated procedure. We proposed a hyperheuristic framework to automate and optimise the choice of the parameters of the GPILS, referred to as HH-GPILS. We experimented with both sequential, namely SA, TS and VNS as well as their hybrids, and parallel high-level methodologies, namely GA.

The study on sequential HH-GPILS showed that a neighbourhood search strategy with the right level of intensification and diversification could lead to optimal or near-optimal solutions. As for the GA-based HH-GPILS, the performance of this hyperheuristic was better than the sequential ones.

We also investigated the influence of the choice of initialising the infeasible solution on the performance of the GPILS. The empirical investigation showed that, in general, initialising the infeasible solution using the *PIH*, could lead to better solutions, in comparison with initialising with AP-based relaxation of the TSP.

Furthermore, we investigated the reusability of the sets of parameters, generated previously, on new problems. The empirical investigation showed that GPILS given previously generated parameters can produce good quality solution for new problem instances. Later, we proposed an offline learning that makes use of knowledge-based system, referred as chromosomes base (CB), where a chromosome is a set of parameters of GPILS. The proposed CB is used to keep track of the best performing sets of parameters, used to solve previous TSP problems, and their scores, which are dependent on their overall performance. The obtained results showed that this learning mechanism used in initialising the population of the GA-based HH-GPILS can produce good quality solutions.

In conclusion, the empirical results show that searching the infeasible space of a COP such as the TSP, which are progressively repaired and locally improved, could lead to the design of promising heuristics, since the infeasible space is larger than the feasible one for any COP. Thus, further efforts should be made to enrich the design features of

local search methods operating in the infeasible solution space and reduce the computational requirements of GPILS and HH-GPILS, on one hand, and further investigations should be made to explore the merit of infeasible search methodologies in solving other COPs, on the other hand.

Potential benefits: this new framework has the potential to renew interest of the academic community in the field of local search methods, on one hand, and allow practitioners to improve the solution frameworks used to address real life applications – especially for online and real time applications where there is a need to repair solutions to decision problems as real life settings changes over time.

6.2. Extensions and Future Work

In this section we shall present future directions and extensions of the thesis:

1. ***Enhancing the proposed GPILS***

   In future research, we shall consider lessons learned from experiments performed in this study. In this thesis, we focused on demonstrating the possibilities of searching in the infeasible space and producing good quality solutions. The empirical investigation in chapter 3 proved that GPILS could produce good quality solutions in a reasonable time. However, when using hyperheuristic to automate its parameters the computational time became unattractive, which is the case even for hyperheuristics proposed for primal search methods. Besides, we shall focus more on the computational time as well as the quality of the solutions produced by GPILS. To do so, we can make use of a choice function that includes the execution time of the LLH (Chen et al, 2016).

2. ***Implementing the proposed GPILS for variants of routing problems as well as other COPs.***

   Since the possibility of heuristically searching in the infeasible space has been proven and GPILS has been developed and tested for TSP, the work should not

stop there. One can explore the infeasible solution space of other COPs, since it is by far larger than the feasible space.

3. ***Improving the offline learning mechanism for GA-based hyperheuristic.*** Being able to reuse the set of parameters of GPILS one can use more advanced offline learning mechanisms followed by online learning mechanisms to enhance the hyperheuristics' performance. For instance, one can intensify the search to parameters in the knowledge-based system; another can diversify the search to parameters not in the knowledge-based system.

6.3. Final remarks

In summary, there are four main contributions of this thesis. Firstly, developing a generic and parameterised local search framework that starts and explores the infeasible space, until it lands into the feasible space. We demonstrated that this new line of research can produce good quality solutions, thus, heuristically searching the infeasible space needs the same attention as searching the feasible and feasible-infeasible space. Secondly, since the proposed framework is parameterised, we proposed hyperheuristic framework to automate the choice of its parameters. We experimented with both sequential; namely, SA, TS, VNS and their hybrids, and parallel based high-level mechanisms. Thirdly, we showed that the reusability of the proposed framework on new and unseen problem is promising. Finally, we proposed an offline learning mechanism that keeps track of previously generated set of parameters and used them to initialise population of the GA-based hyperheuristic

This new framework has the potential to renew interest of the academic community in the field of local search methods, on one hand, and allow practitioners to improve the solution frameworks used to address real life applications – especially for online and real time applications where there is a need to repair solutions to decision problems as real life settings changes over time.

Appendices

Appendix A: Tour construction heuristics

In this section, we only explain some of the tour construction heuristics; namely nearest neighbour construction procedure, insertion procedure, minimal spanning tree procedure, nearest merger procedures and path merging procedures.

The ***Nearest Neighbour Construction Procedure*** starts with any node as the beginning of a path and keeps augmenting that path with the node closest to the last node added to the path until all nodes are included. Finally, the first and the last nodes of the path are joined.

The ***Clarke and Wright savings procedure*** starts with any node k along with back-and-forth routes between node k and any other node in the network. Then, in each iteration, two routes are merged into a single route, where the choice of the two routes to merge is by the magnitude of the savings that would result from the merging operation.

The ***insertion procedure*** starts with any node, say i, finds the closest node to i, say k, and forms the subtour $i \to k \to i$. Then it keeps augmenting that subtour by performing two main steps, namely the selection step and the insertion step, until all nodes are included. In arbitrary insertion, the selection step determines in a random fashion which node k not already in the subtour should join the subtour next. In nearest insertion, the selection step is the minimum distance from any node in the current subtour. However, in farthest insertion the selection step is the maximum distance from any node in the current subtour. In cheapest insertion, the selection step is the minimum insertion cost of the node in the current subtour. The insertion step for all insertion procedures determines where in the subtour the selected node k should be inserted using as an insertion criterion the minimum insertion cost; i.e., the insertion step finds arc (i, j) that minimises $c_{i,k} + c_{k,j} - c_{i,j}$ and inserts between i and j.

Christofides' heuristic (Christofides, 1976) starts with augmenting the set of edges of a minimum spanning tree with edges from the solution to the minimum cost perfect matching on those odd degree nodes of the tree, which leads to a cycle, and transforms the cycle into a hamiltonian cycle by using shortcuts to bypass nodes that appear in the eulerian cycle more than once.

Nearest merger procedures (Rosenkrantz et al., 1977; Glover et al., 2001) starts with n number of subtours with cardinality one. Then, in each iteration, two closest subtours are merged into a single subtour in an optimal way. This process continues until all subtours are merged.

One of the ***path merging procedures*** is recursive path contraction (RPC). RPC starts with the AP-based relaxation of the TSP solution as an initial solution, which typically consists of a number of subtours. Then it repeats the following contract and patch process iteratively; it deletes a most expensive arc in each subtour and contracts the obtained paths, updates the cost matrix with the super-nodes and solves the assignment problem-based relaxation of the TSP on the current set of super-nodes. This iterative process stops when the solution has only one subtour and replaces super-nodes of the TSP tour with the corresponding contracted paths.

Appendix B: Cooling strategies

In this section, some of the popular cooling strategies are presented, such as Aarts and Van Laarhoven (1985, 1987), Lundy and Mees (1986), Huang et al. (1986), Triki et al. (2005), Dowsland (1993) and Azizi and Zolfaghari (2004).

One of the popular cooling strategies is the temperature reduction function is as follows:

$$\alpha(\tau) = \tau_{t-1}/(1 + \alpha \times \tau_{t-1})$$

Aarts and Van Laarhoven (1985a, 1987) proposed a dynamic rule for α. They set different values of α at different epochs which is dependent to the standard deviation of objective function values of neighbouring solutions visited at epoch t, σ_t. For a small a value d greater than zero**,** α at epoch t is as computed as follows:

$$\alpha_t = \frac{ln(1+d)}{3\sigma_{t-1}}$$

Whereas, Lundy and Mees (1986) suggested to set α to a value smaller than $1/U$, where U is an upper bound on $(Z(x_0) - Z^*)$.

Huang et al. (1986) incorporated the expected difference in the average cost at two consecutive epochs, $t-1$ and t, in the cooling schedule. They proposed the following temperature reduction function:

$$\tau_t = \tau_{t-1}.exp\left(-\lambda\tau_{t-1}/\sigma_{t-1}\right)$$

where $\lambda = \Delta/\sigma_{t-1}$ where Δ is the expected difference in the average cost at epoch $t-1$ and t. They suggest using $\lambda = 0.7$.

Later, Triki et al. (2005) proposed a parameter free temperature reduction function that considers the difference in the average cost of two consecutive epochs. They suggest initialising $\Delta(t)$ to value proportional to σ, estimated by a random walk.

$$\tau_t = \tau_{t-1}.\left(1 - \tau_{t-1}\frac{\Delta(t)}{\sigma_{t-1}^2}\right)$$

In cooling strategy, the search starts at a high temperature allowing most of the high hill moves to be accepted and decreases the temperature reducing the probability of acceptance of high-hill moves. Higher probability of acceptance of high-hill moves increases the chance of moving away from local optima at the beginning of the search, and lower probability of acceptance decreases such a chance.

However, adaptive temperature change strategy, cooling and reheating strategy, start with cooling the system, and then the heating is triggered automatically by a prespecified factor and cools the system again. In other words, this strategy changes the probability of acceptance of high-hill moves throughout the process, continuously.

Dowsland (1993) proposed a cooling and reheating strategy that cools the system according to $\tau/1 + \gamma_c\tau$ (Lundy & Mees, 1986) every time a move is accepted, and heats according to $\tau/1 - \gamma_h\tau$ the system every time a move is rejected. If $\gamma_c = \mathcal{K}\gamma_h$, the system will need to go through $\mathcal{K}$ heating iterations to balance one cooling. If the

ratio of rejected to accepted moves is greater than $\boldsymbol{\mathcal{K}}$, then the system heats up; otherwise, the system cools.

On the other hand, Azizi and Zolfaghari (2004) proposed a cooling and reheating schedule where the temperature is controlled by a single function that always maintains τ above a minimum level τ_{min} (e.g. $\tau_{min} = 1$). In this schedule, the heating process gradually takes place if there is any uphill move but the cooling is sudden with the first downhill move.

$$\tau_t = \tau_{min} + \lambda\, ln\big(1 + \rho_i\big), \lambda > 0$$

where λ is a parameter that controls the rate of temperature rise (e.g.,$\lambda = 1$), and ρ_i is the number of consecutive uphill moves at iteration t, ($\rho_0 = 0$).

$$\rho_i = \begin{cases} \rho_{i-1} + 1 & if\ \delta > 0 \\ \rho_{i-1} & if\ \delta = 0 \\ 0 & if\ \delta < 0 \end{cases}$$

Appendix C: Acceptance function

Since, Metropolis criterion, proposed by Kirkpatrick et al. (1983), is dependent on the quality of the current neighbour, Parthasarathy and Rajendran (1997b) proposed setting $\theta = \frac{(\boldsymbol{Z(x_0) - Z(x)) \times 100}}{\boldsymbol{z(x_0)}}$ which is percentage increase over the original solution. This value is dimensionless which one could say θ is independent of the problem specifications.

$$APF(\delta, \tau_t) = \begin{cases} exp\left\{-\frac{\theta}{\tau_t}\right\} & \delta < 0 \\ 1 & \delta \geq 0 \end{cases}$$

Aforementioned APFs are dependent on temperature and the change in cost, meaning higher temperature τ_t and lower change in cost δ results in higher value of APFs; conversely, lower temperature τ_t and higher change in cost δ results in lower value of APFs. In other words, worst solutions have higher probability of acceptance at the beginning of the search; however, with reducing the temperature, probability of acceptance of worst solution decreases.

Ogbu and Smith (1990) proposed an APF independent of temperature and the change in cost that reduces the probability of acceptance geometrically:

$$APF(t) = \begin{cases} APF(1)\ (pfac)^{t-1} & \delta < 0 \\ 1 & \delta \geq 0 \end{cases}$$

where $pfac$ is reduction factor ($pfac < 1$) and $APF(1)$ is the initial APF. These values are predetermined and constant throughout the search.

Dowsland (1993) proposed using constant γ, ($\gamma = 0.33$), in the exponential function to reduce the probability between accepting small and large values of δ. This constant is used to flatten the exponential function.

$$PF(\delta, \tau_t) = \begin{cases} exp\left\{-\left(\frac{\delta}{t+\gamma}\right)\right\} & \delta < 0 \\ 1 & \delta \geq 0 \end{cases}$$

Dueck and Scheuer (1990) and Moscato and Fontanari (1990) proposed a threshold-based acceptance function, which is independent of temperature and quality of the solution. This acceptance function accepts worst solutions only if the increase in cost is less than a threshold. In their method, they used prespecified threshold sequence $Q = \{\mathrm{Q}_1, \mathrm{Q}_2, \dots, \mathrm{Q}_{\mathrm{t_{max}}}\}$.

$$APF(\delta, t) = \begin{cases} 1 & if\ \delta < Q_t \\ 0 & if\ \delta > Q_t \end{cases}$$

These acceptance functions could be classified as either deterministic (e.g., Dueck and Scheuer, 1990; Moscato and Fontanari, 1990; etc.) or stochastic (e.g., Kirkpatrick et al., 1983; Johnson et al., 1989; Brandimarte et al., 1987; etc.).

The aforementioned acceptance functions could be classified as probabilistic and deterministic. Probabilistic acceptance functions are either dependent or independent to current temperature or the change in objective function. As for deterministic acceptance functions, also called Threshold Accepting Algorithms, they are either dependent or independent on the current temperature.

Appendix D: Neighbourhood change strategies

Hansen et al. (2016) classified neighbourhood change strategies as follows:

a. *Sequential neighbourhood change strategy:* Given a specific order of neighbourhood structures, say $NS_k,\ k = 1, \dots, k_{max}$, the sequential strategy continues the search in the next neighbourhood structure until an improvement is achieved, see Table 60. Whenever an improvement occurs, the search will be resumed from the first neighbourhood structure.

$\boldsymbol{Sequential\_Neighborhood\_Change(x, x', k)}$ **{**
IF $z(x') < z(x)$ **THEN**
Update the current seed solution x to the new solution x'; that is, set $x = x'$, and $z(x) = z(x')$;
Set $k = 1$;
ELSE Increment k by 1.
}

Table 60 Sequential neighbourhood change strategy

b. *Cyclic neighbourhood change strategy:* The Cyclic strategy continues the search in the next neighbourhood structure whether an improvement is achieved or not, see Table 61.

Cyclic_$Neighborhood_Change(x, x', k)$ **{**
IF $z(x') < z(x)$ **THEN**
Update the current seed solution; that is, set $x = \mathrm{x}'$, and $z(x) = z(x')$;
Increment k by 1.
}

Table 61 Cyclic neighbourhood change strategy

c. *Pipe neighbourhood change strategy:* The search in every neighbourhood is continued until no improvement is achieved, see Table 62.

$\boldsymbol{Pipe\_Neighborhood\_Change(x, x', k)}$ **{**
IF $z(x') < z(x)$ **THEN**
Update the current seed solution; that is, set $x = \mathrm{x}'$, and $z(x) = z(x')$;
ELSE Increment k by 1.
}

Table 62 Pipe neighbourhood change strategy

d. *Skewed neighbourhood change strategy:* A neighbourhood change strategy may accept uphill moves with some ratio, see Table 63. If the ratio includes the difference between the values of the objective functions, this

neighbourhood change strategy is called Skewed neighbourhood change strategy. In this strategy could be integrated with sequential, cyclic or pipe neighbourhood change strategy.

Skewed_Neighborhood_Change (x, x', k) **{**
IF $z(x') - z(x) < \alpha . d(x', x)$ **THEN**
Update the current seed solution; that is, set $x = x'$, and $z(x) = z(x')$;
Set $k = 1$;
ELSE Increment k by 1.
}

Table 63 Skewed neighbourhood change strategy

Where $d(x', x)$ is the distance between x and x'.

Appendix E: GA's selection mechanisms

In this section, the GA's selection mechanisms are explained; namely ordinal selection, proportional selection, ranking selection, steady-state selection.

Ordinal selection

According to this selection mechanism, the chance a chromosome is selected to be a parent is based on its rank (order) in comparison with others in the current generation. The most common ordinal selection mechanisms are *tournament selection* and *truncation selection.*

Tournament selection (Goldberg et al. 1989): a number of chromosomes from the current population, say s, are selected at random. These chromosomes compete and the fittest chromosome in the group wins the tournament. In order to choose n chromosomes, n tournaments are required.

Truncation selection (Mühlenbein and Schlierkamp-Voosen, 1993): only select a fraction of the fittest chromosomes. In this selection mechanism a threshold, say T, is specified. This threshold indicates the fraction of the population to be selected.

Proportional selection

Proportional selection is a probability-based selection according to fitness value. Two commonly used proportional selections are *roulette- wheel selection* and *stochastic universal selection (SUS).*

Roulette- wheel selection, see Figure 40, is a probability-based selection according to fitness value. Roulette wheel selection method assigns a slot to each chromosome with probability p_i, where p_i is proportional to the chromosome's fitness and is calculated by the following formula:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

A single chromosome is selected by spinning the roulette wheel, thus, to select a set of chromosomes, it should be done multiple times. Furthermore, the roulette wheel gives a higher chance of being chosen to chromosomes with higher probability.

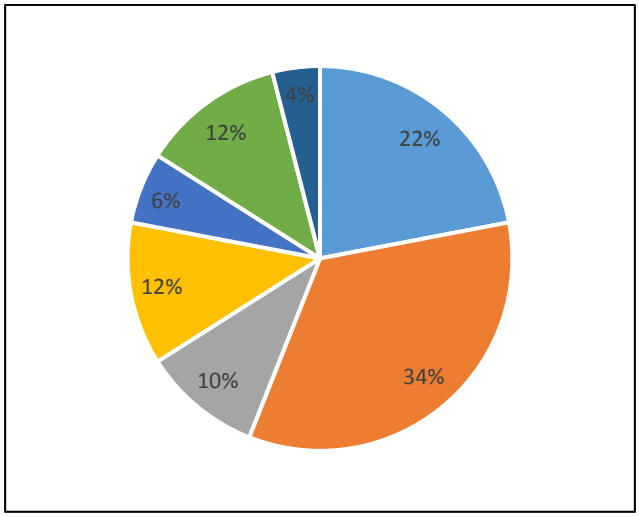


Figure 40 Roulette- wheel selection

Stochastic universal selection (SUS) (Baker, 1987) is an unbiased variation of the roulette wheel that selects a number of needed chromosomes by a single spinning of the roulette wheel, see Figure 41. In this selection mechanism, a set of evenly spaced markers is placed outside the roulette wheel. The roulette wheel is turned only once. Selected chromosomes are the ones that the markers have fallen on their slot. Although this selection gives a fair chance to weaker chromosomes to be chosen, if a chromosome has a big slot in the wheel, SUS performance declines.

Ranking selection

In proportional selection, if some of the chromosomes have a big slot in the roulette wheel, they will have higher chance to be chosen that leads to fast and premature convergence. To overcome the beforementioned issue, a rank-based fitness assignment (Baker, 1985) has been proposed, giving all chromosomes a chance to be chosen. Ranking selection convergence is slower than proportional selection.

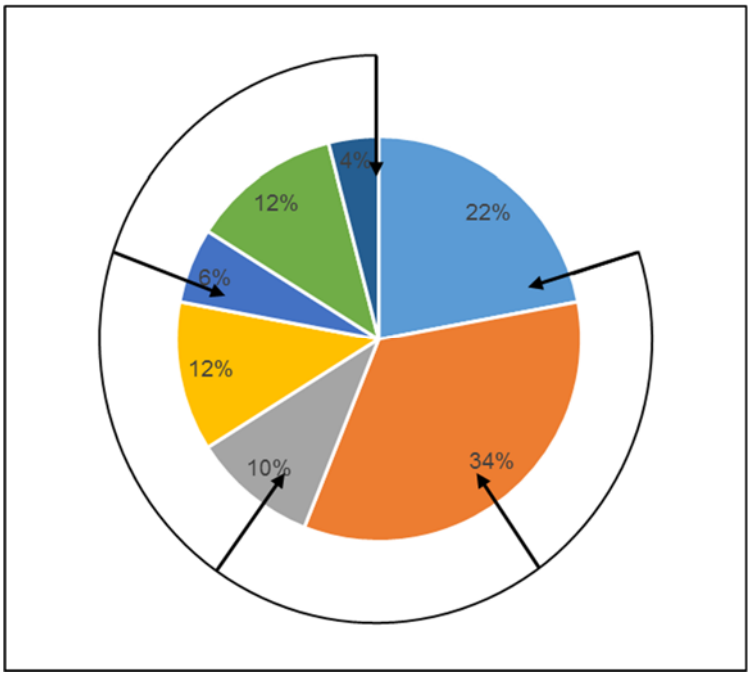


Figure 41 Stochastic universal selection

In this selection mechanism, chromosomes are sorted based on their fitness, from best to worst, and rank them. Then the selection probability, say $Prob_i$, assigned to each chromosome i depends on its rank r_i and not its actual fitness value. Then a proportionate selection according to these probabilities is performed.

$$Rrob_i = \frac{r_i}{\sum_{j=1}^{n} r_j}$$

Steady-state selection

Steady-state or genitor selection chooses two chromosomes; the first is for reproduction and the second chromosome is to be replaced by the new offspring. The choice of the chromosome for reproduction is made by a linear ranking method and the selection of the chromosome to be replaced is the worst in the current population.

Appendix F: Crossover techniques

Several crossover techniques have been proposed to combine the parents' genes to generate new offspring such as simple or one-point crossover, multi-point crossover, uniform crossover and three parents' crossover.

Simple or One-point crossover: this operator is the simplest and the most common crossover operator. As it is shown in Figure 42, a crossover point on both chromosomes is randomly chosen, and all the alleles after the crossover point are exchanged.

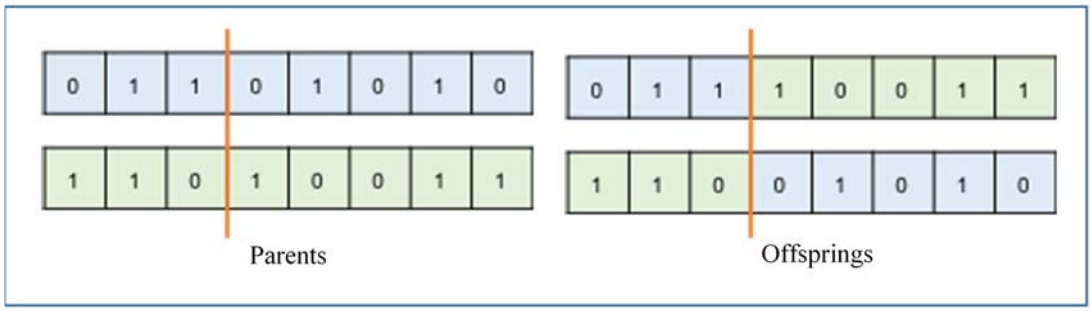


Figure 42 One-point crossover

Multi-point crossover: another common crossover is multipoint crossover; the two-point is mostly used amongst them. In the two-point crossover, two crossover points are randomly chosen, and the segment between the two are exchanged, see Figure 43. Multi-point Crossover follows the same concept as one-point and two-point crossover.

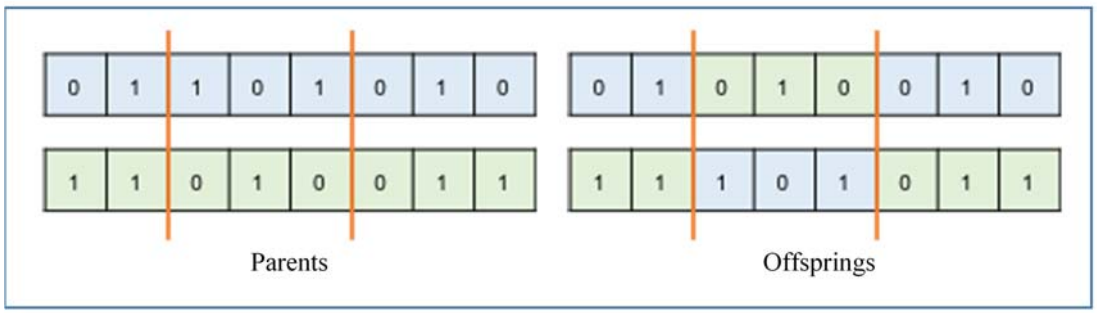


Figure 43 Two-point crossover

Uniform Crossover: one-point and two-point crossovers' contribution to the offsprings are only segments of genes. One might argue that this crossover mechanism is not exploratory enough. Uniform crossover (Syswerda, 1989; Spears and De Jong, 1995), recombines the parents in genes level, diversifying the search while inheriting the

parents' genes and information. In this crossover mechanism, parents' alleles are randomly swapped with probability p_{uc}, see Figure 44.

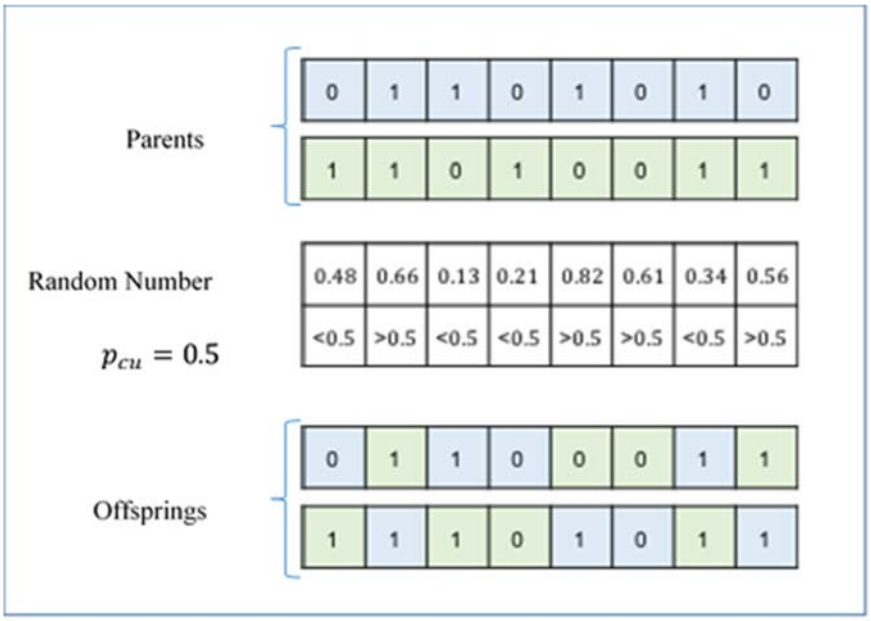


Figure 44 Uniform crossover

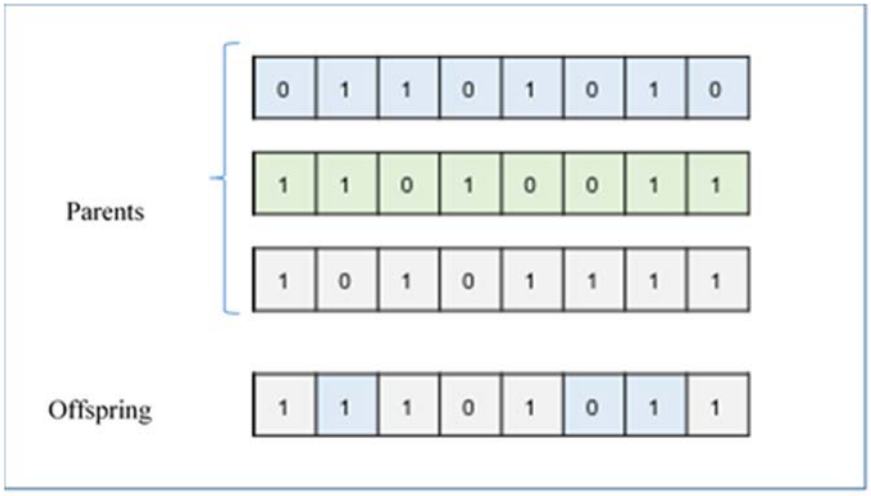


Figure 45 Three parents' crossover

Three parents' crossover (Sivanandam and Deepa, 2007): In this crossover technique, three parents are randomly chosen and compared. Each allele of the first parent is compared with the allele of the second parent. If both are the same, the allele is taken for the offspring; otherwise, the allele from the third parent is taken for the offspring. This concept is illustrated in Figure 45.

Appendix G: Hyperheuristic classifications and categories

Soubeiga (2003), see Figure 46, first classified hyperheuristics in two categories, namely single-heuristic and multiple heuristics. In the former, a single Parameterised heuristic is used to solve a problem. A hyperheuristic is developed to find the optimal or near optimal set of parameters. However, the latter (multiple heuristics) deal with more than one heuristic and assist in the search for the best heuristic or set of heuristics to solve the problem.

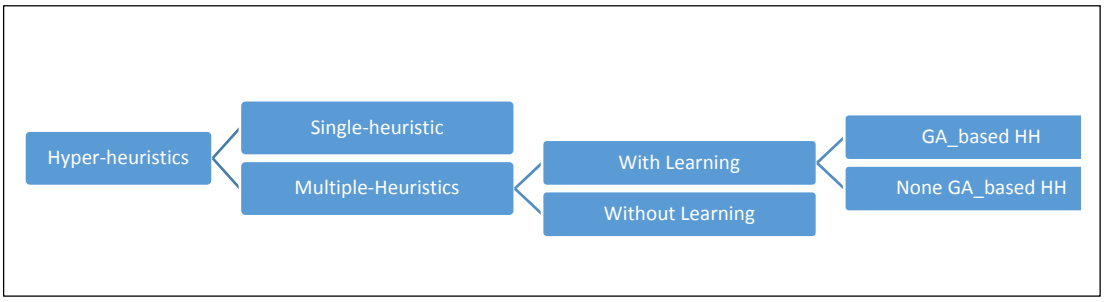


Figure 46 Soubeiga (2003) hyperheuritic classification

Soubeiga (2003) further classified the second category, multiple heuristics, based on learning, hyperheuristics without learning and hyperheuristics with learning. Learning methods gather historical data about the performance of the system and use learning mechanisms to improve the performance of the system. In the context of hyperheuristic algorithms, learning methods collect knowledge concerning the performance of heuristics or components in the HH search space to select the best performing (set of) heuristic(s) or component(s) using a learning mechanism. hyperheuristics without learning select the next low-level heuristic or neighbourhood structure based on a predetermined sequence. On the other hand, hyperheuristics with learning make use of a learning mechanism to select the next low-level heuristic or neighbourhood structure. In their classification, they divided the learning mechanisms into GA-based hyperheuristics and other hyperheuristics with learning.

In another classification by Bai (2005) and Ross (2005), hyperheuristics are categorised as constructive hyperheuristic and perturbative (local search methods) hyperheuristic, Figure 47. Constructive Hyperheuristic, given combinatorial problem and a set of constructive low-level heuristics, constructs the solution from scratch

incrementally. The hyperheuristic stops when a complete solution is achieved. On the other hand, perturbative hyperheuristic, given an initial complete solution and perturbative low-level heuristics leads the search to promising neighbourhoods.

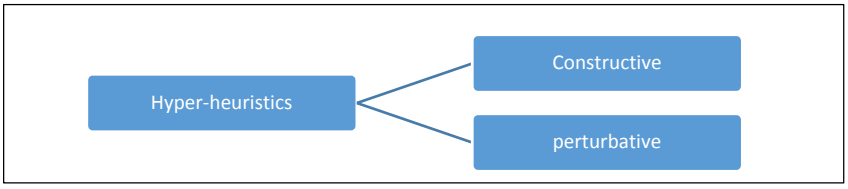


Figure 47 Bai (2005) and Ross (2005) classification

In another classification mentioned independently by Burke et al. (2010), Bader-El-Den, and Poli (2007), hyperheuristics were classified into two groups, namely heuristic selection and heuristic generation, see Figure 48. Heuristic selection is "heuristic to select heuristic". In this category of hyperheuristics, given a set of low-level heuristics, search the heuristic space to find the best solution. On the other hand, heuristic generation is a "heuristic to generate heuristic". Given a set of components or building blocks, this hyperheuristic selects the best configuration to produce a new heuristic to solve the problem at hand. Furthermore, this newly generated heuristic could be used to address other problems (usable heuristic) or it could not (disposable heuristic).

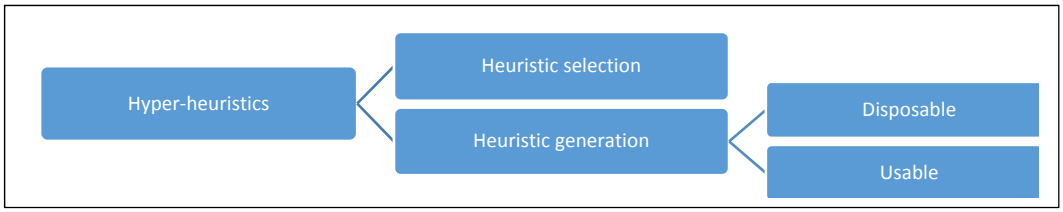


Figure 48 Hyperheuristic classification of Burke et al. (2010), Bader-El-Den, and Poli (2007)

Moreover, Burke et al. (2010, and 2013) considered the most fundamental classification represented by heuristic selection and heuristic generation and defined hyperheuristic as an automated method to select or generate heuristic to solve COPs. They classified hyperheuristics based on two dimensions, the first nature of the search space and the second source of feedback through learning. The first level of the nature of the search space is dependent on whether hyperheuristic is designed to select heuristic, amongst given set of heuristics, or generate heuristic, given the set of components. Furthermore, the second level makes a distinction between perturbative and constructive heuristics, Figure 49.

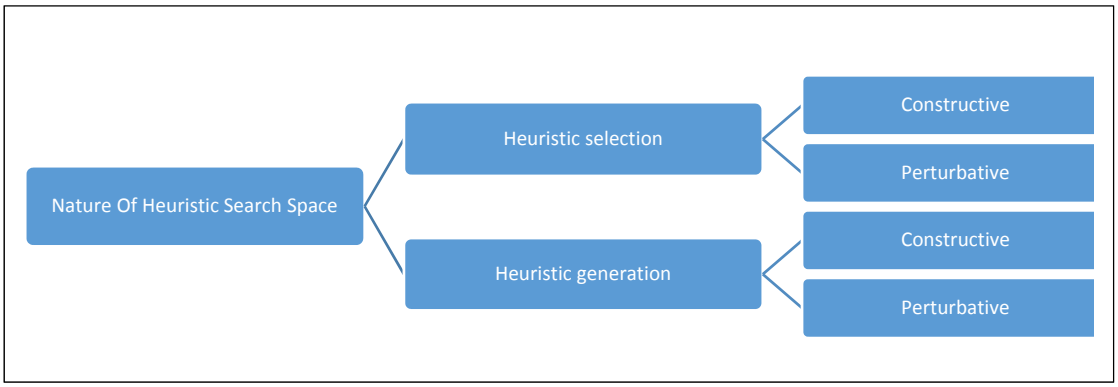


Figure 49 Hyperheuristic's first dimension

The second dimension is learning based, with learning and without learning. Learning could be either online or offline, Figure 50. Hyperheuristic with online learning, hyperheuristic collects knowledge about the system during the process. However, hyperheuristic with offline learning, hyperheuristic collects knowledge about the system from a set of training instances, in the form of rules.

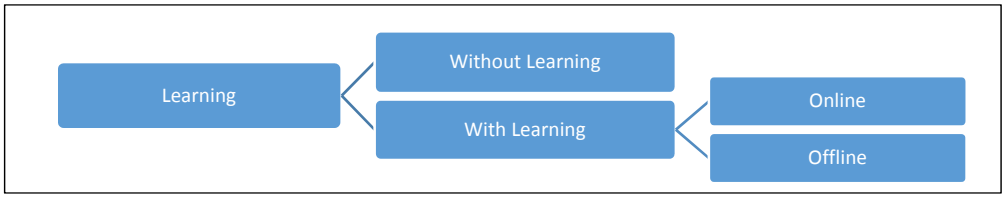


Figure 50 Hyperheuristics second dimension

Appendix H: Learning mechanisms

Choice Function (CH) is well-known in multi-criteria decision-making. CH is a statistical ranking of alternatives that guide the search for the best choice of alternative or set of alternatives based on the provided historical knowledge about their performance throughout the time.

We first explain the idea behind choice function: consider a system with a set of alternatives, where each alternative has a set of criterions. The system should choose a single or a set of alternatives at time t that lead to its optimal or close to optimal performance. Choice function evaluates performance of each alternative i at time t as follows:

$$CF_t(A_i) = \sum_{j=1}^{NC} \alpha_j f_{it}(c_j)$$

where NC indicates number of criterions of the alternatives, c_j indicates criterion j of the alternative, $f_{it}(c_j)$ is the score of c_j at time t and α_j is a weight indicating the importance of criterion j in the choice function. The system selects alternative i with the highest value of $CF_t(A_i)$, at time $t+1$.

The main components in designing choice function are weights of the criterion, α_j. These weights should be chosen correctly. The choice of these weights requires a warmup phase.

In the context of hyperheuristics, the set of alternatives could be low-level heuristics or components, which in this section they are referred to as LLH. Moreover, Criteria considered for calculation of CF could be the gathered historical data about their recent effectiveness or performance; such as the time needed to perform LLH, the last time LLH has been called, etc. Selection mechanisms used in exploration could be roulette wheel (proportional to $\frac{CF(A_i)}{\sum_j CF(A_j)}$), maximum value (highest CF value), rank-based selection (ranking LLH based on their performance), etc. (Cowling et al., 2001; Chen et al., 2016). Cowling et al. (2000) introduced a choice function to select the next LLH to be called. The proposed CF measures the effectiveness LLHs based on the current provided historical knowledge about the exploited LLH search space. They considered three criterions to update the efficiency of each LLH_i; namely, information regarding recent its effectiveness, recent effectiveness of consecutive pairs of LLH and the time since it was last called. They experimented with four different selection mechanisms; namely, roulette wheel, maximum value, rank-based selection and $DecompChoice$ (consider LLH producing best scores for each criterion and CF). Their results suggest that CF- based hyperheuristic, which accept non-improving LLH, is significantly better than hyperheuristic without learning. In addition, they observed that $DecompChoice$ selection mechanism performs better than other selection mechanisms, and roulette wheel is performing better than the maximum value, rank-based selection.

Later, Chen et al. (2016) used a different CF. They updated $CF(LLH_i)$ by considering the following three criteria: performance of LLH_i estimated by both fitness change and execution time, performance of collaboration of LLHs in pairs (estimated by

successively applying pairs of LLH) and the time since $LLH_i$was last called. Their proposed selection mechanism was $w\%$ of LLHs with highest rank.

Reinforcement Learning (RL) is a reward-based mechanism, which provides feedback in terms of reward and penalty, based on the system's performance over time. RL is an online learning mechanism that interacts with the environment and gathers information, called exploitation process, and uses the gathered information to select the next action to take, called exploration process. Since the environment might be unknown, a trial and error is needed to gather information and explore the environment. In designing an adequate RL, a trade-off should be made between exploration and exploitation.

In the context of hyperheuristic, an improving LLH will be rewarded by increasing its weight; otherwise, it will be penalised by decreasing its weight. A Selection mechanism is used to select an LLH based on their weight, such as maximum weight or roulette wheel (Nareyek, 2003; Ozcan et al., 2010; Chen et al., 2016). Nareyek (2003) used RL to select promising heuristics at each decision point. They evaluated different variants of selection mechanisms and weight adaptation. They considered maximum weight and roulette wheel as selection mechanism, and the following reward and penalty schemes to update the weights (w_h):

Reward Schemes:

R_1 (Additive adaptation): $w_h \leftarrow w_h + 1$

R_2 (Escalating additive adaptation): $w_h \leftarrow w_h + reward$

R_3 (Multiplicative adaptation): $w_h \leftarrow w_h \times 2$

R_4 (Escalating multiplicative adaptation): $w_h \leftarrow w_h \times Reward$

R_5 (Power adaptation): $w_h \leftarrow \begin{cases} w_h^2 : w_h > 1 \\ 2 \quad : w_h = 1 \end{cases}$

Penalty Schemes:

P_1 (Subtractive adaptation): $w_h \leftarrow w_h - 1$

P_2 (Escalating Subtractive adaptation): $w_h \leftarrow w_h - Penalty$

P_3 (Divisional adaptation): $w_h \leftarrow w_h/2$

P_4 (Escalating divisional adaptation): $w_h \leftarrow w_h/Penalty$

P_5 (Root adaptation): $w_h \leftarrow \sqrt{w_h}$

Any combination of weight adaptation (reward and penalty scheme) and selection mechanism could be used to design a reinforcement mechanism. Their analysis suggests that a small reward (R_1) in case of an improvement and large penalty (P_5) in case of deterioration is a good combination of weight adaptation. In addition, a selection mechanism based on maximum weight is often a better exploration strategy in comparison with selection mechanism based on roulette wheel.

Learning Classifier System (LCS) is a rule-based machine learning method that identifies set (population) of rules, representing knowledge about the environment, learns and evolves the population iteratively to make predictions. LCS was first reported by Holland and Reitman (1978). Learning Classifier Systems are a combination of two components. The first is a discovery component which identifies set of rules (if: then conditions), which are not known yet, and the second is a learning component that uses the accumulated knowledge about the environment to guide the discovery component to improve its performance. The Discovery component is an evolutionary algorithm, typically GA, and learning component can be reinforcement learning, also known as credit assignment (Holland and Reitman; 1978). Ross et al. (2002) proposed a hyperheuristic for Bin-packing problem, which combines a set of LLHs. They proposed a learning classifier system that evolves condition-action rules to learn which LLH to call in each decision point. See also Ross (2005) and Marín-Blázquez and Schulenburg (2007).

Case-Based Reasoning (CBR) is based on two principles of nature, similar problems have similar solutions and it is more likely that future problem might be like the current ones (Leake, 1996). Considering these two principles, to solve a new problem (new case), instead of starting from scratch, one can retrieve similar experienced problem situations (cases) and adapt them to solve the new one in hand. CBR is a knowledge-based technique that stores experienced cases in a memory (case base) when it faces a new case retrieves similar cases, using a similarity measure, and adapts previous

experience to the new case. Regardless of failure or success, CBR learns from the new experience and revise its general knowledge to exploit previous success and avoid future failures. A critical factor in the success of CBR is case representation in the case base. A case usually consists of a representation of the problem, its features and conditions of retrieval, and the solution (Burke et al., 2002). Burke et al. (2002) proposed a hyperheuristic using CBR to solve timetabling problems. The purpose of using CBR was to predict the best LLH to address new problems by retrieving old cases. See also (Petrovic and Qu, 2002, Burke et al. 2002, 2004, 2006).

Appendix I: $\mathcal{K}$-Means clustering

$\mathcal{K}$-Means is an iterative cluster enhancement technique, see Table 64. This iterative procedure starts with an initial set of centroids, usually chosen at random, and alternates between two steps; namely assigning nodes to clusters and updating centroids (MacKay, 2003). This process continues until the clusters converge, meaning that no change in their centroids has been observed. Note that this algorithm is almost surely converges after finite number of iterations (Bottou and Bengio, 1995).

Initialization steps

Place $\mathcal{K}$ centroids $C_1, \ldots, C_{\mathcal{K}}$ at random locations;

Iterative steps

REPEAT until the convergence criterion is met

  FOR each node i **{**

    Compute the distance from i to each centroid;

    Sort centroids in decreasing distance from node i;

    Assign customer i to the cluster whose centroid is the closest;

  }

  Update centroid coordinates based on the current assignment scheme by computing each centroids' coordinates as the mean of the coordinates of the customers assigned to that cluster;

  Check whether the convergence criterion is met or not;

END REPEAT

Table 64 $\mathcal{K}$-means clustering algorithm

This technique is easy to implement and apply on large data sets. It has been used in several applications such as signal processing, cluster analysis, feature learning etc.

In this thesis, we used this clustering method to exploit the structure of TSP instances, e.g. the distribution of nodes being random, clustered or uniform.

Appendix J: RINS function

A recursive function iteratively calls itself and shows the output at the end of each iteration (Butterfield et al., 2016). It is used when solving a problem requires solving a smaller or different version of the same problem. For example, consider the calculation of the factorial of a natural number:

$$n! = n \times (n-1) \times (n-2) \ldots \times 1$$

For example:

$$3! = 3 \times 2 \times 1$$

$$4! = 4 \times 3!$$

$$5! = 5 \times 4!$$

Therefore, $n!$ can be rewritten as follows:

$$n! = n \times (n-1)!$$

In other words, for calculating $n!$, one can simply solve smaller sub-problem, $((n-1)!)$ and multiply it by n. Thus, the factorial function can be designed either iteratively using FOR loops or recursively using a recursive function, see Table 65. This procedure is a common method used in computer programming since it allows the programmer to write an efficient and generic code, since one can implement finite number of codes for any number of recursion.

Iterative	**Recursive**

Factorial (n) {
$Result = \boldsymbol{n}$;
IF ($n = 0$)
$Result = 1$;
ELSE IF ($\boldsymbol{n > 0}$) {
FOR ($i = n-1; i > 0; i--$){
$Result \times= i$;
}
}
return $Result$;
}
	Factorial (n) {
	IF ($n = 0$)
	return 1;
	ELSE
	return $n \times$ Factorial$(n-1)$;
	}

Table 65 Pseudo-code of recursive versus iterative factorial function

As it was mentioned in chapter 3, exploring the infeasible space for a given set of parameters requires exploring a single or several combinations of s and r. One can use an iterative design, where for any combination of (s, r) several FOR loops are required. For example, for combinations of $r = 1$ arc in $s = 2$ subtours, two embedded FOR loops are needed; however, for combinations of $r = 2$ arc in $s = 3$ subtours, six embedded FOR loops are needed. As a result, the iterative design restricts us to limited combinations of (s, r) as it requires a different code for each combination.

Therefore, to make the code efficient and generic; i.e., any combination of (s, r) could be implemented with the same code, we proposed a recursive function to search the infeasible neighbourhood, see Table 24 for a detailed pseudo-code of (RINS). As it was mentioned earlier the recursive function iteratively calls itself and shows the output at the end of each iterations, see Table 24. An example with $s^k = 2$ and $\{r_1^k = 2, r_2^k = 1\}$ is illustrated in Figure 51.

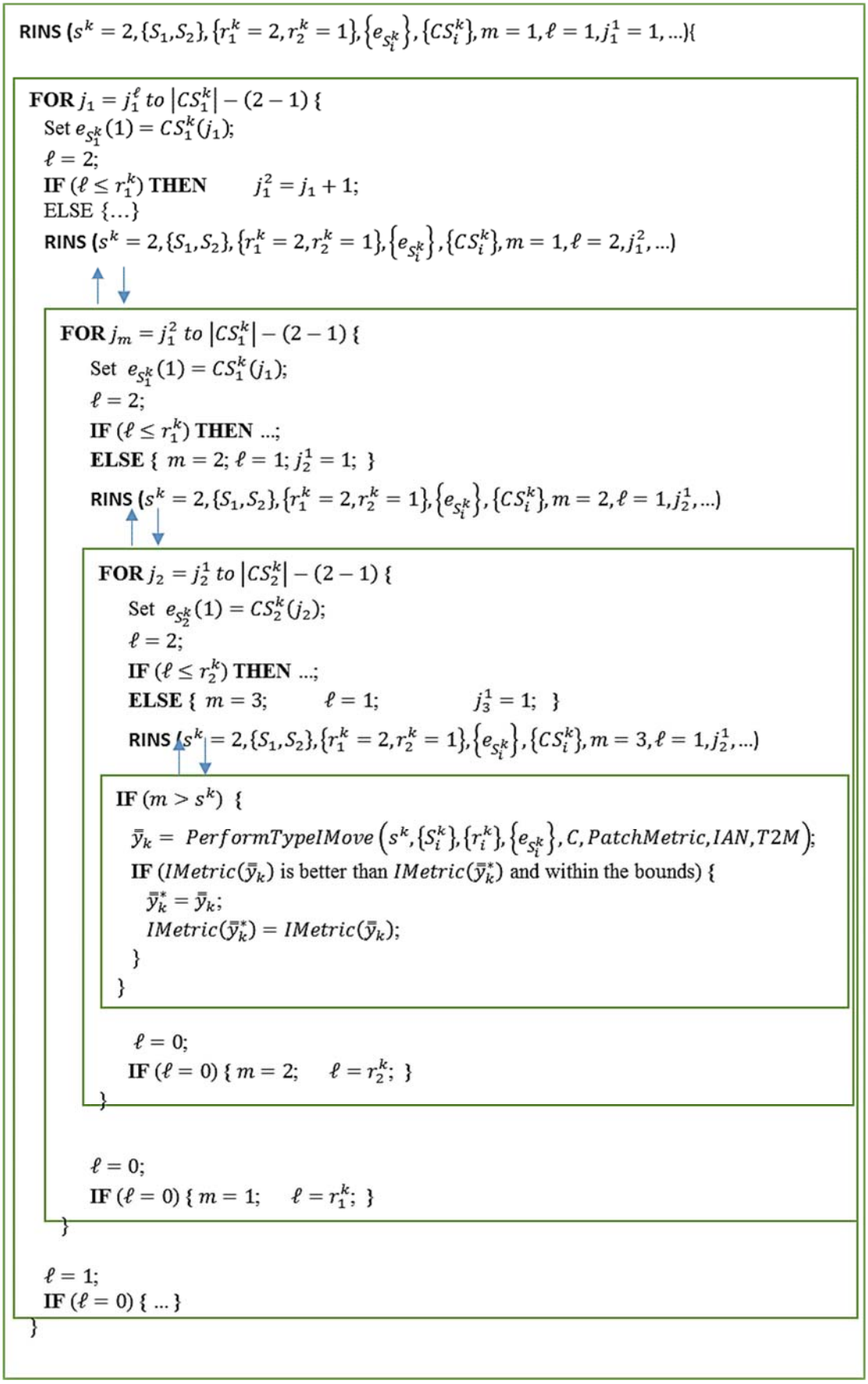


Figure 51 RINS example

References

Aarts, Emile HL, and Peter JM Van Laarhoven. 1985. "Statistical cooling: A general approach to combinatorial optimization problems." Philips J. Res. 40 (4):193-226.

Alander, Jarmo T. 1992. "On optimal population size of genetic algorithms." CompEuro'92.'Computer Systems and Software Engineering', Proceedings.

Applegate, David, Robert Bixby, William Cook, and Vasek Chvátal. 1998. "On the solution of traveling salesman problems."

Applegate, David L, Robert E Bixby, Vasek Chvatal, and William J Cook. 2006. The traveling salesman problem: a computational study: Princeton University press.

Archetti, Claudia, Maria Grazia Speranza, and Alain Hertz. 2006. "A tabu search algorithm for the split delivery vehicle routing problem." Transportation science 40 (1):64-73.

Arostegui, Marvin A., Sukran N. Kadipasaoglu, and Basheer M. Khumawala. 2006. "An empirical comparison of Tabu Search, Simulated Annealing, and Genetic Algorithms for facilities location problems." International Journal of Production Economics 103 (2):742-754. doi: https://doi.org/10.1016/j.ijpe.2005.08.010.

Atiqullah, Mir M. 2004. "An efficient simple cooling schedule for simulated annealing." International Conference on Computational Science and Its Applications.

Avella, Pasquale, Maurizio Boccia, and Laurence A Wolsey. 2017. "Single-period cutting planes for inventory routing problems." Transportation Science 52 (3):497-508.

Aytug, Haldun, and Gary J Koehler. 1996. "Stopping criteria for finite length genetic algorithms." INFORMS Journal on Computing 8 (2):183-191.

Azizi, Nader, and Saeed Zolfaghari. 2004. "Adaptive temperature control for simulated annealing: a comparative study." Computers & Operations Research 31 (14):2439-2451. doi: http://dx.doi.org/10.1016/S0305-0548(03)00197-7.

Bäck, Thomas. 1992. "The interaction of mutation-rate, selection, and self-adaptation within a genetic algorithm." Männer and Manderick 1503:85-94.

Bäck, Thomas. 1996. Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, NY.

Bäck, Thomas, and Martin Schütz. 1996. "Intelligent mutation rate control in canonical genetic algorithms." In Foundations of Intelligent Systems: 9th International Symposium, ISMIS '96 Zakopane, Poland, June 9–13, 1996 Proceedings, edited by Zbigniew W. Raś and Maciek Michalewicz, 158-167. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bader-El-Den, Mohamed, and Riccardo Poli. 2007. "Generating SAT local-search heuristics using a GP hyper-heuristic framework." Artificial evolution.

Bai, Ruibin. 2005. "An investigation of novel approaches for optimising retail shelf space allocation." University of Nottingham.

Bai, Ruibin, and Graham Kendall. 2005. "An investigation of automated planograms using a simulated annealing based hyper-heuristic." In Metaheuristics: Progress as real problem solvers, 87-108. Springer.

Baker, James E. 1987. "Reducing bias and inefficiency in the selection algorithm." Proceedings of the second international conference on genetic algorithms.

Balas, Egon, and Nicos Christofides. 1981. "A restricted Lagrangean approach to the traveling salesman problem." Mathematical Programming 21 (1):19-46.

Battiti, R., and G. Tecchiolli. 1994. "Simulated annealing and Tabu search in the long run: A comparison on QAP tasks." Computers and Mathematics with Applications 28 (6):1-8. doi: 10.1016/0898-1221(94)00147-2.

Bellmore, Mandell, and John C Malone. 1971. "Pathology of traveling-salesman subtour-elimination algorithms." Operations Research 19 (2):278-307.

Bock, F. 1958. "An algorithm for solving traveling-salesman andrelated network optimization problems." Unpublishedmanuscript associated with talk presented at the 14th ORSANational Meeting.

Bottou, Leon, and Yoshua Bengio. 1995. "Convergence properties of the k-means algorithms." Advances in neural information processing systems.

Brandimarte, P, R Conterno, and P Laface. 1987. "FMS production scheduling by simulated annealing." Proceedings of the third International Conference on Simulation in Manufacturing.

Burke, E, and E Soubeiga. 2003. "Scheduling nurses using a tabu-search hyperheuristic." Proceedings of the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2003), Nottingham, UK.

Burke, E. K., B. L. MacCarthy, S. Petrovic, and R. Qu. 2003. Knowledge discovery in a hyper-heuristic for course timetabling using case-based reasoning. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Burke, Edmund, Moshe Dror, Sanja Petrovic, and Rong Qu. 2005. "Hybrid graph heuristics within a hyper-heuristic approach to exam timetabling problems." In The next wave in computing, optimization, and decision technologies, 79-91. Springer.

Burke, Edmund K, Peter I Cowling, and Ralf Keuthen. 2001. "Effective local and guided variable neighbourhood search methods for the asymmetric travelling salesman problem." Workshops on Applications of Evolutionary Computation.

Burke, Edmund K, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. "Hyper-heuristics: A survey of the state of the art." Journal of the Operational Research Society 64 (12):1695-1724.

Burke, Edmund K, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. 2009. "Exploring hyper-heuristic methodologies with genetic programming." In Computational intelligence, 177-201. Springer.

Burke, Edmund K, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. 2010. "A classification of hyper-heuristic approaches." In Handbook of metaheuristics, 449-468. Springer.

Burke, Edmund K, Graham Kendall, Mustafa Mısır, Ender Özcan, EK Burke, G Kendall, E Özcan, and M Mısır. 2004. "Applications to timetabling." Handbook of Graph Theory, chapter 5.6.

Burke, Edmund K, Bart L MacCarthy, Sanja Petrovic, and Rong Qu. 2002. "Knowledge discovery in a hyper-heuristic for course timetabling using case-based reasoning." International Conference on the Practice and Theory of Automated Timetabling.

Burke, Edmund K, Sanja Petrovic, and Rong Qu. 2006. "Case-based heuristic selection for timetabling problems." Journal of Scheduling 9 (2):115-132.

Butterfield, Andrew, Gerard Ekembe Ngondi, and Anne Kerr. 2016. A dictionary of Computer Science: Oxford University Press.

Chakhlevitch, Konstantin, and Peter Cowling. 2008. "Hyperheuristics: Recent Developments." In Adaptive and Multilevel Metaheuristics, edited by Carlos Cotta, Marc Sevaux and Kenneth Sörensen, 3-29. Berlin, Heidelberg: Springer Berlin Heidelberg.

Chen, Yujie, Philip Mourdjis, Fiona Polack, Peter Cowling, and Stephen Remde. 2016. "Evaluating hyperheuristics and local search operators for periodic routing problems." European Conference on Evolutionary Computation in Combinatorial Optimization.

Chiang, W. C., and C. Chiang. 1998. "Intelligent local search strategies for solving facility layout problems with the quadratic assignment problem formulation." European Journal of Operational Research 106 (2-3):457-488. doi: 10.1016/S0377-2217(97)00285-3.

Christofides, Nicos. 1970. "The shortest Hamiltonian chain of a graph." SIAM Journal on Applied Mathematics 19 (4):689-696.

Christofides, Nicos. 1975a. Graph Theory: An algorithmic approach: New York: Academic Press Inc.

Christofides, Nicos. 1975b. "Hamiltonian circuits and the travelling salesman problem." In Combinatorial Programming: Methods and Applications, 149-171. Springer.

Christofides, Nicos. 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

Clarke, Geoff, and John W Wright. 1964. "Scheduling of vehicles from a central depot to a number of delivery points." Operations research 12 (4):568-581.

Connolly, David. 1992. "General purpose simulated annealing." Journal of the Operational Research Society 43 (5):495-505.

Connolly, David T. 1990. "An improved annealing scheme for the QAP." European Journal of Operational Research 46 (1):93-100.

Cordeau, Jean-François, Gilbert Laporte, and Anne Mercier. 2001. "A unified tabu search heuristic for vehicle routing problems with time windows." Journal of the Operational research society 52 (8):928-936.

Cordeau, Jean‐François, Michel Gendreau, and Gilbert Laporte. 1997. "A tabu search heuristic for periodic and multi‐depot vehicle routing problems." Networks 30 (2):105-119.

Cowling, Peter, Graham Kendall, and Limin Han. 2002. "An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem." Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on.

Cowling, Peter, Graham Kendall, and Eric Soubeiga. 2000. "A hyperheuristic approach to scheduling a sales summit." In Practice and theory of automated timetabling III, 176-190. Springer.

Cowling, Peter, Graham Kendall, and Eric Soubeiga. 2001. "A parameter-free hyperheuristic for scheduling a sales summit." Proceedings of the 4th metaheuristic international conference.

Cowling, Peter, Graham Kendall, and Eric Soubeiga. 2002. "Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation." In Applications of evolutionary computing, 1-10. Springer.

Crainic, Teodor G., Michel Gendreau, Patrick Soriano, and Michel Toulouse. 1993. "A tabu search procedure for multicommodity location/allocation with balancing requirements." Annals of Operations Research 41 (4):359-383. doi: 10.1007/bf02023001.

Croes, Georges A. 1958. "A method for solving traveling-salesman problems." Operations research 6 (6):791-812.

Crowder, Harlan, and Manfred W Padberg. 1980. "Solving large-scale symmetric travelling salesman problems to optimality." Management Science 26 (5):495-509.

Cuervo, Daniel Palhazi, Peter Goos, Kenneth Sörensen, and Emely Arráiz. 2014. "An iterated local search algorithm for the vehicle routing problem with backhauls." European Journal of Operational Research 237 (2):454-464.

Dantzig, George, Ray Fulkerson, and Selmer Johnson. 1954. "Solution of a large-scale traveling-salesman problem." Journal of the operations research society of America 2 (4):393-410.

Daridi, F, N Kharma, and J Salik. 2004. "Parameterless genetic algorithms: review and innovation." IEEE Canadian Review 47:19-23.

De Jong, Kenneth. 1988. "Learning with genetic algorithms: An overview." Machine learning 3 (2):121-138.

de Lucena Filho, Abilio Pereira. 1986. "Exact solution approaches for the vehicle routing problem." Ph.D. Thesis, Department of Management Science, Imperial College of Science and Technology, University of London, Amsterdam

Denzinger, Jorg, Marc Fuchs, and Matthias Fuchs. 1997. "High performance ATP systems by combining several AI methods." Proceedings of the 15th international joint conference on Artifical intelligence-Volume 1.

DePuy, Gail W, Reinaldo J Moraga, and Gary E Whitehouse. 2005. "Meta-RaPS: a simple and effective approach for solving the traveling salesman problem." Transportation Research Part E: Logistics and Transportation Review 41 (2):115-130.

Dowsland, Kathryn A. 1993. "Some experiments with simulated annealing techniques for packing problems." European Journal of Operational Research 68 (3):389-399.

Dueck, Gunter, and Tobias Scheuer. 1990. "Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing." Journal of computational physics 90 (1):161-175.

Eastman, Willard Lawrence. 1958. "Linear programming with pattern constraints: a thesis." Harvard University.

Finke, Gerd. 1984. "A two-commodity network flow approach to the traveling salesman problem." Congresses Numeration 41:167-178.

Fischetti, Matteo, Andrea Lodi, and Paolo Toth. 2003. "Solving real-world ATSP instances by branch-and-cut." In Combinatorial Optimization—Eureka, You Shrink!, 64-77. Springer.

Fischetti, Matteo, and Paolo Toth. 1997. "A polyhedral approach to the asymmetric traveling salesman problem." Management Science 43 (11):1520-1536.

Fisher, H., and G.L. Thompson. 1963. "Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules." Prentice-Hall, Englewood Cliffs:225-251.

Fix, Evelyn, and Joseph L Hodges Jr. 1951. Discriminatory analysis-nonparametric discrimination: consistency properties. California Univ Berkeley.

Flood, Merrill M. 1956. "The Traveling-Salesman Problem." Operations Research 4 (1):61-75.

Fogarty, Terence C. 1989. "An incremental genetic algorithm for real-time optimisation." Systems, Man and Cybernetics, 1989. Conference Proceedings., IEEE International Conference.

Fox, Kenneth R, Bezalel Gavish, and Stephen C Graves. 1980. "An n-constraint formulation of the (time-dependent) traveling salesman problem." Operations Research 28 (4):1018-1021.

Friden, Charles, Alain Hertz, and D de Werra. 1989. "Stabulus: A technique for finding stable sets in large graphs with tabu search." Computing 42 (1):35-44.

Gavish, Bezalel, and Stephen C Graves. 1978. "The travelling salesman problem and related problems."

Gavish, Bezalel, and K Srikanth. 1983. Efficient branch and bound code for solving large scale travelling salesman problems to optimality: University of Rochester. Graduate School of Management.

Gendreau, Michel, Alain Hertz, and Gilbert Laporte. 1992. "New insertion and postoptimization procedures for the traveling salesman problem." Operations Research 40 (6):1086-1094.

Gendreau, Michel, Alain Hertz, and Gilbert Laporte. 1994. "A tabu search heuristic for the vehicle routing problem." Management science 40 (10):1276-1290.

Gendreau, Michel, and Jean-Yves Potvin. 2014. "Tabu Search." In Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, edited by Edmund K. Burke and Graham Kendall, 243-263. Boston, MA: Springer US.

Gendreau, Michel, Patrick Soriano, and Louis Salvail. 1993. "Solving the maximum clique problem using a tabu search approach." Annals of Operations Research 41 (4):385-403.

Geng, Xiutang, Zhihua Chen, Wei Yang, Deqian Shi, and Kai Zhao. 2011. "Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search." Applied Soft Computing 11 (4):3680-3689.

Glover, Fred. 1977. "Heuristics for integer programming using surrogate constraints." Decision Sciences 8 (1):156-166. doi: 10.1111/j.1540-5915.1977.tb01074.x.

Glover, Fred. 1986. "Future paths for integer programming and links to artificial intelligence." Computers & operations research 13 (5):533-549.

Glover, Fred. 1989. "Tabu search—part I." ORSA Journal on computing 1 (3):190-206.

Glover, Fred. 1990. "Tabu search—part II." ORSA Journal on computing 2 (1):4-32.

Glover, Fred. 1992. "New ejection chain and alternating path methods for traveling salesman problems." In Computer science and operations research, 491-509. Elsevier.

Glover, Fred. 1996. "Ejection chains, reference structures and alternating path methods for traveling salesman problems." Discrete Applied Mathematics 65 (1-3):223-253.

Glover, Fred, Gregory Gutin, Anders Yeo, and Alexey Zverovich. 2001. "Construction heuristics for the asymmetric TSP." European Journal of Operational Research 129 (3):555-568. doi: http://dx.doi.org/10.1016/S0377-2217(99)00468-3.

Glover, Fred, and Manuel Laguna. 1997. "General purpose heuristics for integer programming—Part I." Journal of Heuristics 2 (4):343-358.

Glover, Fred W, and Gary A Kochenberger. 2006. Handbook of metaheuristics. Vol. 57: Springer Science & Business Media.

Goldberg, DE. 1989. "Genetic algorithms in search, optimization, and machine learning, addison-wesley, reading, ma, 1989." NN Schraudolph and J 3 (1).

Golden, B., L. Bodin, T. Doyle, and W. Stewart. 1980. "Approximate Traveling Salesman Algorithms." Operations Research 28 (3):694-711.

Gomory, Ralph E. 1958. "Outline of an algorithm for integer solutions to linear programs." Bulletin of the American Mathematical society 64 (5):275-278.

Grefenstette, J. J. 1986. "Optimization of control parameters for genetic algorithms." IEEE Transactions on Systems, Man, and Cybernetics 16 (1):122-128. doi: 10.1109/TSMC.1986.289288.

Hansen, Pierre, and Nenad Mladenovic. 2003. A tutorial on variable neighborhood search: Groupe d'études et de recherche en analyse des décisions, HEC Montréal.

Hansen, Pierre, and Nenad Mladenović. 2006. "First vs. best improvement: an empirical study." Discrete Applied Mathematics 154 (5):802-817.

Hansen, Pierre, Nenad Mladenović, Raca Todosijević, and Saïd Hanafi. 2016. "Variable neighborhood search: basics and variants." EURO Journal on Computational Optimization:1-32.

Hart, Emma, Peter Ross, and Jeremy Nelson. 1998. "Solving a real-world problem using an evolving heuristically driven schedule builder." Evolutionary Computation 6 (1):61-80.

Hart, Emma, Peter Ross, and Jeremy AD Nelson. 1999. "Scheduling chicken catching - An investigationinto the success of a genetic algorithm on areal - world scheduling problem." Annals of Operations Research 92:363-380.

Hartigan, John A, and Manchek A Wong. 1979. "Algorithm AS 136: A k-means clustering algorithm." Journal of the Royal Statistical Society. Series C (Applied Statistics) 28 (1):100-108.

He, Yi, Yuhui Qiu, Guangyuan Liu, and Kaiyou Lei. 2005. "A parallel adaptive tabu search approach for traveling salesman problems." Natural Language Processing and Knowledge Engineering, 2005. IEEE NLP-KE'05. Proceedings of 2005 IEEE International Conference.

Held, Michael, and Richard M Karp. 1970. "The traveling-salesman problem and minimum spanning trees." Operations Research 18 (6):1138-1162.

Held, Michael, and Richard M Karp. 1971. "The traveling-salesman problem and minimum spanning trees: Part II." Mathematical programming 1 (1):6-25.

Helsgaun, Keld. 2000. "An effective implementation of the Lin–Kernighan traveling salesman heuristic." European Journal of Operational Research 126 (1):106-130.

Hertz, Alain, and Dominique de Werra. 1987. "Using tabu search techniques for graph coloring." Computing 39 (4):345-351.

Hesser, Jürgen, and Reinhard Männer. 1991. "Towards an optimal mutation probability for genetic algorithms." Parallel Problem Solving from Nature:23-32.

Ho, Sin C, and Dag Haugland. 2004. "A tabu search heuristic for the vehicle routing problem with time windows and split deliveries." Computers & Operations Research 31 (12):1947-1964.

Holland, JH. 1989. Genetic algorithms in search, optimization and machine learning. Addison-Wesley, Reading, MA.

Holland, John H, and Judith S Reitman. 1978. "Cognitive systems based on adaptive algorithms." In Pattern-directed inference systems, 313-329. Elsevier.

Holland, John H. 1975. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence: U Michigan Press.

Houck, David J 1978. The traveling salesman problem as a constrained shortest path problem: Theory and computational experience: Ecole polytechnique de Montréal.

Huang, YJ, Hashem Akbari, Haider Taha, and Arthur H Rosenfeld. 1987. "The potential of vegetation in reducing summer cooling loads in residential buildings." Journal of climate and Applied Meteorology 26 (9):1103-1116.

Hussin, Mohamed Saifullah , and Thomas Stützle. 2014. "Tabu search vs. simulated annealing as a function of the size of quadratic assignment problem instances." Computers & Operations Research 43:286-291. doi: https://doi.org/10.1016/j.cor.2013.10.007.

Jain, Anil K. 2010. "Data clustering: 50 years beyond K-means." Pattern recognition letters 31 (8):651-666.

Johnson, David S, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. 1989. "Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning." Operations research 37 (6):865-892.

Kanungo, Tapas, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. "An efficient k-means clustering algorithm:

Analysis and implementation." IEEE transactions on pattern analysis and machine intelligence 24 (7):881-892.

Karp, Richard M. 1977. "Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane." Mathematics of operations research 2 (3):209-224.

Katayama, K, H Sakamoto, and H Narihisa. 2000. "The efficiency of hybrid mutation genetic algorithm for the travelling salesman problem." Mathematical and Computer Modelling 31 (10-12):197-203.

Katayama, Kengo, Masafumi Tani, and Hiroyuki Narihisa. 2000. "Solving large binary quadratic programming problems by effective genetic local search algorithm." Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation.

Kaur, D, and MM Murugappan. 2008. "Performance enhancement in solving traveling salesman problem using hybrid genetic algorithm." Fuzzy Information Processing Society, 2008. NAFIPS 2008. Annual Meeting of the North American.

Kim, CE. 1975. A minimal spanning tree and approximate tours for a traveling salesman: University of Maryland. Computer Science.

Kirkpatrick, Scott. 1984. "Optimization by simulated annealing: Quantitative studies." Journal of statistical physics 34 (5-6):975-986.

Kirkpatrick, Scott, C Daniel Gelatt, and Mario P Vecchi. 1983. "Optimization by simmulated annealing." science 220 (4598):671-680.

Koulamas, C., S. R. Antony, and R. Jaen. 1994. "A survey of simulated annealing applications to operations research problems." Omega 22 (1):41-56. doi: http://dx.doi.org/10.1016/0305-0483(94)90006-X.

Koulinas, G., L. Kotsikas, and K. Anagnostopoulos. 2014. "A particle swarm optimization based hyper-heuristic algorithm for the classic resource constrained project scheduling problem." Information Sciences 277:680-693. doi: 10.1016/j.ins.2014.02.155.

Langevin, A. 1988. "Planification de tournées de véhicules."

Laporte, Gilbert. 1992. "The traveling salesman problem: An overview of exact and approximate algorithms." European Journal of Operational Research 59 (2):231-247.

LAWLER, EL. 1976. "Combinatorial Optimization: Networks and Matroids." Holt, Rinehart and Winston.

Leake, David B. 1996. Case-Based Reasoning: Experiences, Lessons and Future Directions: MIT Press.

Liao, Xiao-Ping. 2009. "An orthogonal genetic algorithm with total flowtime minimization for the no-wait flow shop problem." Machine Learning and Cybernetics, 2009 International Conference.

Lin, Shen. 1965. "Computer solutions of the traveling salesman problem." Bell System Technical Journal 44 (10):2245-2269.

Lin, Shen, and Brian W Kernighan. 1973. "An effective heuristic algorithm for the traveling-salesman problem." Operations research 21 (2):498-516.

Little, John DC, Katta G Murty, Dura W Sweeney, and Caroline Karel. 1963. "An algorithm for the traveling salesman problem." Operations research 11 (6):972-989.

Loulou, R.J. 1988. "On multicommodity flow formulation for the TSP."

Lourenço, Helena R, Olivier C Martin, and Thomas Stutzle. 2003. "Iterated local search." International series in operations research and management science:321-354.

Lundy, M. 1985. "Applications of the annealing algorithm to combinatorial problems in statistics." Biometrika 72 (1):191-198.

Lundy, Miranda, and Alistair Mees. 1986. "Convergence of an annealing algorithm." Mathematical programming 34 (1):111-124.

MacKay, David JC, and David JC Mac Kay. 2003. Information theory, inference and learning algorithms: Cambridge university press.

Mak, King-Tim, and Andrew J Morton. 1993. "A modified Lin-Kernighan traveling-salesman heuristic." Operations Research Letters 13 (3):127-132.

Marín-Blázquez, Javier G., and Sonia Schulenburg. 2007. "A Hyper-Heuristic Framework with XCS: Learning to Create Novel Problem-Solving Algorithms Constructed from Simpler Algorithmic Ingredients." In Learning Classifier Systems: International Workshops, IWLCS 2003-2005, Revised Selected Papers, edited by Tim Kovacs, Xavier Llorà, Keiki Takadama, Pier Luca Lanzi, Wolfgang Stolzmann and Stewart W. Wilson, 193-218. Berlin, Heidelberg: Springer Berlin Heidelberg.

McCall, John. 2005. "Genetic algorithms for modelling and optimisation." Journal of Computational and Applied Mathematics 184 (1):205-222.

Miliotis, P. 1978. "Using cutting planes to solve the symmetric travelling salesman problem." Mathematical programming 15 (1):177-188.

Miller, C. E., A. W. Tucker, and R. A. Zemlin. 1960. "Integer Programming Formulation of Traveling Salesman Problems." J. ACM 7 (4):326-329. doi: 10.1145/321043.321046.

MILLER, DONALDL, and JOSEPHF PEKNY. 1991. "Exact solution of large asymmetric traveling salesman problems." Science 251 (4995):754-761.

Mirkin, Gerri, Kris Vasudevan, Frederick A. Cook, William G. Laidlaw, and William G. Wilson. 1993. "A comparison of several cooling schedules for simulated annealing implemented on a residual statics problem." Geophysical Research Letters 20 (1):77-80. doi: 10.1029/92GL03024.

Mladenović, Nenad, and Pierre Hansen. 1997. "Variable neighborhood search." Computers & operations research 24 (11):1097-1100.

Montané, Fermín Alfredo Tang, and Roberto Diéguez Galvao. 2006. "A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service." Computers & Operations Research 33 (3):595-619.

Moscato, Pablo, and José F Fontanari. 1990. "Stochastic versus deterministic update in simulated annealing." Physics Letters A 146 (4):204-208.

Mühlenbein, Heinz, and Dirk Schlierkamp-Voosen. 1993. "Predictive models for the breeder genetic algorithm i. continuous parameter optimization." Evolutionary computation 1 (1):25-49.

Mumford, Christine L. 2004. "Simple population replacement strategies for a steady-state multi-objective evolutionary algorithm." Genetic and Evolutionary Computation Conference.

Nareyek, Alexander. 2003. "Choosing search heuristics by non-stationary reinforcement learning." In Metaheuristics: Computer decision-making, 523-544. Springer.

Nourani, Yaghout, and Bjarne Andresen. 1998. "A comparison of simulated annealing cooling strategies." Journal of Physics A: Mathematical and General 31 (41):8373.

Ogbu, F. A., and D. K. Smith. 1990. "The application of the simulated annealing algorithm to the solution of the n/m/Cmax flowshop problem." Computers & Operations Research 17 (3):243-253. doi: http://dx.doi.org/10.1016/0305-0548(90)90001-N.

Or, I. 1976. "Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking." PhD thesis (Department of Industrial Engineering and Management Science, Northwestern University).

Orman, A.J., and H.P. Williams. 2007. "A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem." Berlin, Heidelberg.

Ouenniche, Jamal, Prasanna K Ramaswamy, and Michel Gendreau. 2017. "A dual local search framework for combinatorial optimization problems with TSP application." Journal of the Operational Research Society:1-22.

Özcan, Ender, Mustafa Misir, Gabriela Ochoa, and Edmund K Burke. 2012. "A Reinforcement Learning: Great-Deluge Hyper-Heuristic for Examination Timetabling." In Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends, 34-55. IGI Global.

Padberg, Manfred, and Giovanni Rinaldi. 1987. "Optimization of a 532-city symmetric traveling salesman problem by branch and cut." Operations Research Letters 6 (1):1-7.

Parthasarathy, S, and Chandrasekharan Rajendran. 1997a. "A simulated annealing heuristic for scheduling to minimize mean weighted tardiness in a flowshop with sequence-dependent setup times of jobs-a case study." Production Planning & Control 8 (5):475-483.

Parthasarathy, Srinivasaraghavan, and Chandrasekharan Rajendran. 1997b. "An experimental evaluation of heuristics for scheduling in a real-life flowshop with sequence-dependent setup times of jobs." International journal of production economics 49 (3):255-263.

Paul, P Victer, N Moganarangan, S Sampath Kumar, R Raju, T Vengattaraman, and P Dhavachelvan. 2015. "Performance analyses over population seeding techniques of the permutation-coded genetic algorithm: An empirical study based on traveling salesman problems." Applied Soft Computing 32:383-402.

Paul, P. V., P. Dhavachelvan, and R. Baskaran. 2013. "A novel population initialization technique for Genetic Algorithm." 2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT), 20-21 March 2013.

Paulli, J. 1993. "A computational comparison of simulated annealing and tabu search applied to the quadratic assignment problem." In Applied Simulated Annealing, 85-102. Springer.

Petrovic, Sanja, and Rong Qu. 2002. "Case-based reasoning as a heuristic selector in a hyper-heuristic for course timetabling problems." Proceedings of the Sixth International Conference on Knowledge-Based Intelligent Information & Engineering Systems (KES'2002), Crema, Italy 336-340.

Potvin, Jean-Yves, Guy Lapalme, and Jean-Marc Rousseau. 1989. "A generalized k-opt exchange procedure for the MTSP." INFOR: Information Systems and Operational Research 27 (4):474-481.

Puchinger, Jakob, and Günther R. Raidl. 2005. "Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification." In Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005, Las Palmas, Canary Islands, Spain, June 15-18, 2005, Proceedings, Part II, edited by José Mira and José R. Álvarez, 41-53. Berlin, Heidelberg: Springer Berlin Heidelberg.

Qu, Liangsheng, and Ruixiang Sun. 1999. "A synergetic approach to genetic algorithms for solving traveling salesman problem." Information Sciences 117 (3):267-283.

Qu, Rong, and Edmund Burke. 2005. "Hybrid variable neighborhood hyperheuristics for exam timetabling problems." The Sixth Metaheuristics International Conference 2005, Aug, 2005, Vienna, Austria.

Ray, Shubhra Sankar, Sanghamitra Bandyopadhyay, and Sankar K Pal. 2007. "Genetic operators for combinatorial optimization in TSP and microarray gene ordering." Applied intelligence 26 (3):183-195.

Rego, Cesar. 1998. "A subpath ejection method for the vehicle routing problem." Management Science 44 (10):1447-1459.

Reinelt, Gerhard. 1994. The traveling salesman: computational solutions for TSP applications: Springer-Verlag.

Renaud, Jacques, Gilbert Laporte, and Fayez F Boctor. 1996. "A tabu search heuristic for the multi-depot vehicle routing problem." Computers & Operations Research 23 (3):229-235.

Roeva, O., S. Fidanova, and M. Paprzycki. 2013. "Influence of the population size on the genetic algorithm performance in case of cultivation process modelling." 2013 Federated Conference on Computer Science and Information Systems, 8-11 Sept. 2013.

Rosenkrantz, Daniel J, Richard E Stearns, and II Lewis, Philip M. 1977. "An analysis of several heuristics for the traveling salesman problem." SIAM journal on computing 6 (3):563-581.

Ross, Hsiao-Lan Fang1and Peter, and Dave Corne. 1994. "A promising hybrid GA/heuristic approach for open-shop scheduling problems." Proc. 11< sup> th</sup> European Conference on Artificial Intelligence.

Ross, Peter. 2005. "Hyper-heuristics." In Search methodologies, 529-556. Springer.

Ross, Peter, Sonia Schulenburg, Javier G Marín-Bläzquez, and Emma Hart. 2002. "Hyper-heuristics: learning to combine simple heuristics in bin-packing problems." Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation.

Russell, Robert A, and Dave Gribbin. 1991. "A multiphase approach to the period routing problem." Networks 21 (7):747-765.

Safe, Martín, Jessica Carballido, Ignacio Ponzoni, and Nélida Brignole. 2004. "On stopping criteria for genetic algorithms." Brazilian Symposium on Artificial Intelligence.

Salhi, Saïd. 2017. Heuristic search: The emerging science of problem solving: Springer.

Schaffer, J David, and Amy Morishima. 1987. "An adaptive crossover distribution mechanism for genetic algorithms." Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms.

Schrimpf, Gerhard, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. 2000. "Record Breaking Optimization Results Using the Ruin and Recreate Principle." Journal of Computational Physics 159 (2):139-171. doi: http://dx.doi.org/10.1006/jcph.1999.6413.

Shanmugam, M, MS Saleem Basha, P Victer Paul, P Dhavachelvan, and R Baskaran. 2013. "Performance assessment over heuristic population seeding techniques of genetic algorithm: benchmark analyses on traveling salesman problems."

International Journal of Applied Engineering Research (IJAER), Research India Publications 8 (10):1171-1184.

Shapiro, Donald M. 1966. "Algorithms for the solution of the optimal cost and bottle-neck traveling salesman problems." Washington University, St. Louis (Thesis).

Sinclair, Marius. 1993. "Comparison of the performance of modern heuristics for combinatorial optimization on real data." Computers & Operations Research 20 (7):687-695. doi: https://doi.org/10.1016/0305-0548(93)90056-O.

Sivanandam, SN, and SN Deepa. 2007. Introduction to genetic algorithms: Springer Science & Business Media.

Skorin-Kapov, Jadranka. 1990. "Tabu search applied to the quadratic assignment problem." ORSA Journal on computing 2 (1):33-45.

Smith, Jim, and Frank Vavak. 1999. "Replacement strategies in steady state genetic algorithms: Static environments." Foundations of genetic algorithms 5:219-233.

Smith, Theunis HC, V Srinivasan, and GL Thompson. 1977. "Computational performance of three subtour elimination algorithms for solving asymmetric traveling salesman problems." Annals of Discrete Mathematics 1:495-506.

Smith, Theunis HC, and Gerald Luther Thompson. 1977. "A LIFO implicit enumeration search algorithm for the symmetric traveling salesman problem using Held and Karp's 1-tree relaxation." Annals of Discrete Mathematics 1:479-493.

Sörensen, Kenneth, and Fred W Glover. 2013. "Metaheuristics." In Encyclopedia of operations research and management science, 960-970. Springer.

Soubeiga, Eric. 2003. "Development and application of hyperheuristics to personnel scheduling." University of Nottingham.

Spears, William M, and Kenneth D De Jong. 1995. On the virtues of parameterized uniform crossover. NAVAL RESEARCH LAB WASHINGTON DC.

Srinivas, M., and L. M. Patnaik. 1994. "Adaptive probabilities of crossover and mutation in genetic algorithms." IEEE Transactions on Systems, Man, and Cybernetics 24 (4):656-667. doi: 10.1109/21.286385.

Steinhöfel, Kathleen, A Albrecht, and CK Wong. 1998. "On various cooling schedules for simulated annealing applied to the job shop problem." International Workshop on Randomization and Approximation Techniques in Computer Science.

Stützle, Thomas. 1998. "Applying iterated local search to the permutation flow shop problem." FG Intellektik, TU Darmstadt, Darmstadt, Germany.

Syswerda, Gilbert. 1989. "Uniform crossover in genetic algorithms." Proceedings of the third international conference on Genetic algorithms.

Taillard, Eric. 1990. "Some efficient heuristic methods for the flow shop sequencing problem." European journal of Operational research 47 (1):65-74.

Taillard, Éric. 1991. "Robust taboo search for the quadratic assignment problem." Parallel computing 17 (4-5):443-455.

Triki, Eric, Yann Collette, and Patrick Siarry. 2005. "A theoretical study on the behavior of simulated annealing leading to a new cooling schedule." European Journal of Operational Research 166 (1):77-92.

Vajda, S. 1961. "Mathematical programming." Mathematical Proceedings.

Van Laarhoven, Peter JM, and Emile HL Aarts. 1987. "Simulated annealing." In Simulated Annealing: Theory and Applications, 7-15. Springer.

Volgenant, Ton, and Roy Jonker. 1982. "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation." European Journal of Operational Research 9 (1):83-89.

Wei, Yingzi, Yulan Hu, and Kanfeng Gu. 2007. "Parallel search strategies for TSPs using a greedy genetic algorithm." Natural Computation, 2007. ICNC 2007. Third International Conference.

Wong, Richard T. 1980. "Integer programming formulations of the traveling salesman problem." Proceedings of the IEEE international conference of circuits and computers.

Yang, Jinhui, Chunguo Wu, Heow Pueh Lee, and Yanchun Liang. 2008. "Solving traveling salesman problems using generalized chromosome genetic algorithm." Progress in Natural Science 18 (7):887-892.

Yeo, A. 1997. "Large exponential neighbourhoods for the TSP." preprint, Dept of Maths and CS, Odense University, Odense, Denmark.

Yugay, Olga, Insoo Kim, Beomjune Kim, and Franz IS Ko. 2008. "Hybrid genetic algorithm for solving traveling salesman problem with sorted population." Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference.